MINING MASSIVE MOVING OBJECT DATASETS FROM RFID DATA FLOW ANALYSIS TO TRAFFIC MINING

BY

HECTOR GONZALEZ

B.S., Universidad Eafit, 1995
M.B.A., Harvard University, 1999
M.S., University of Illinois at Urbana-Champaign, 2003

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Doctoral Committee:

Professor Jiawei Han, Chair
Professor Marianne Winslett
Assistant Professor ChengXiang Zhai
Assistant Professor Magdalena Balazinska

Abstract

Effective management of moving object data, originating in supply chain operations, road network monitoring, and other RFID applications, is a major challenge facing society today, with important implications into business optimization, city planning, privacy, and national security. Towards the solution of this problem, I have developed a comprehensive framework for warehousing, mining, and cleaning large moving object data sets.

The proposed framework addresses the following key challenges present in object tracking applications: (1) Datasets are massive, a single large retailer may generate terabytes of moving object data per day. (2) Data is usually dirty, many tags are not detected at all, or are incorrectly detected at the wrong location. (3) Dimensionality is very large, there are spatio-temporal dimensions defined by object trajectories, sensor related dimensions such as temperature or humidity recorded at different locations, and item level dimensions describing the attributes of each object. (4) Data analysis and mining need to navigate and discover interesting patterns, at different levels of abstraction, and involving a large number of interrelated records in multiple datasets.

At the core of my dissertation, is the RFID data warehousing engine. It receives clean data from the cleaning engine, and provides highly compressed data, at multiple levels of abstraction, to the mining engine. The mining engine is composed of three modules. The first, mines commodity flow patterns that identify general flow trends and significant flow exceptions in a large supply chain operation. The second, makes route recommendations, based on observed driving behavior and traffic conditions. And the third, discovers and characterizes a wide variety of traffic anomalies on a road network.

RFID Data Warehousing. A data warehouse is an enterprise level data repository that collects and integrates organizational data in order to provide decision support analysis. At the core of the data warehouse is the data cube, which computes an aggregate measure (*e.g.*, sum, avg, count) for all possible combination of dimensions of a fact table (*e.g.*, sales for 2004, in the northeast). Online analytical processing (OLAP) operations provide the means for exploration and analysis of the data cube. My research on this direction has extended the data cube to handle moving object data sets [42], by significantly compressing such data, and proposing a new aggregation mechanism that preserves its path structure. The RFID warehouse is built around the concept of the *movement graph*, which records both spatio-temporal and item level information in a compact model. We show that compression and query processing efficiency

can be significantly improved, by partitioning the *movement graph* around gateway nodes, which are special locations connecting different spatial regions in the graph.

RFID Data Cleaning. Efficient and accurate data cleaning is an essential task for the successful deployment of applications, such as object tracking and inventory management systems, based on RFID technology. Most existing data cleaning approaches do not consider the overall cost of cleaning in an environment that possibly includes thousands of readers and millions of tags. We propose a cleaning framework that takes an RFID data set and a collection of cleaning methods, with associated costs, and induces a *cleaning plan* that optimizes the overall *accuracy-adjusted cleaning costs*. The *cleaning plan* determines the conditions under which inexpensive cleaning methods can be safely applied, the conditions under which more expensive methods are absolutely necessary, and those cases when a combination of several methods is the optimal policy. Through a variety of experiments we show that our framework can achieve better accuracy at a fraction of the cost than that obtained by applying any single technique.

Mining Flow Trends. An important application of moving objects is mining movement patterns of objects in supply chain operations. In this context, one may ask questions regarding correlations between time spent at quality control locations and laptop return rates, salient characteristics of dairy products discarded from stores, or ships that spent abnormally long at intermediate ports before arrival. The gigantic size of such data, and the diversity of queries over flow patterns pose great challenges to traditional workflow induction and analysis technologies since processing may involve retrieval and reasoning over a large number of inter-related tuples through different stages of object movements. Creating a complete workflow that records all possible commodity movements and that incorporates time will be prohibitively expensive since there can be billions of different location and time combinations. I propose the FlowGraph [41], as a compressed probabilistic workflow, that captures the general flow trends and significant exceptions of a data set. The FlowGraph achieves compression by recording the set of major flow trends, and the set of non-redundant flow exceptions (*i.e.*, abnormal transitions or durations) present in the data. I extended the concept of the FlowGraph to incorporate multiple levels of abstraction of object and path characteristics, and defined the FlowCube [40], which is a data cube that records FlowGraphs as measures, and that allows OLAP reasoning on object flows.

Mining Route Recommendations. Modern highway networks provide several mechanisms for automatic vehicle identification. The most common are the use of toll collection transponders to detect vehicles at multiple points in the network, and the use of cameras to automatically identify license plates. Such information provides valuable patterns useful to online navigation systems and route planning applications. Most existing route planning applications use a fastest path algorithm based on static or dynamic models of road speeds, but such models in general disregard observed driver behavior, and other important factors such as weather, car-pool availability, or vehicle type. Existing solutions may, for example, provide a route that is the fastest one, but that goes through a high crime area, and is thus avoided by

experienced drivers. We propose a traffic-mining-based path-finding method [43] that mines speed and driving models from historic traffic data, and uses them to compute fast routes that are well supported by historic driving behavior under the set of relevant driving and traffic conditions.

Mining Traffic Anomalies. Identification and characterization of traffic anomalies on massive road networks is a vital component of traffic monitoring [44]. Anomaly identification can be used to reduce congestion, increase safety, and provide transportation engineers with better information for traffic forecasting and road network design. However, due to the size, complexity and dynamics of such transportation networks, it is challenging to automate the process. We propose a multi-dimensional mining framework that can be used to identify a concise set of anomalies from massive traffic monitoring data, and further overlay, contrast, and explore such anomalies in multi-dimensional space.

To my family.

Acknowledgements

This dissertation would not have been possible without the support of many people. Most of them not named here, please accept my apologies.

First, I would like to express my deep gratitude to my thesis advisor, Professor Jiawei Han, for his constant support and encouragement, for his excitement about research, for sharing his thoughts and experiences, and for his patience even during difficult times. Working with him has been a lifetime experience that I will never forget.

I also wish to thank the members of my dissertation committee, Professor ChengXiang Zhai, Professor Marianne Winslett, and Professor Magdalena Balazinska for their feedback and support. They helped me put my work into perspective, and gave me valuable advice not only on academics but also on my personal life.

I would like to thank my collaborators in the DAIS group: Xiaolei Li, Xuehua Shen, and Hong Cheng. Some of the work we have done together appears in this thesis. I am grateful for our discussions and experiences doing research together.

I also thank Professor Diego Klabjan and Professor Yanfeng Ouyang for our collaborations. Our discussions on supply chain management and traffic engineering gave me inspiration and insights that helped me improve and advance many ideas that appear in this thesis.

Finally, I would like to extend my thankfulness to my family for their support throughout all these years. Without your patience and love I would have not survived the difficult times.

Table of Contents

List of Figures

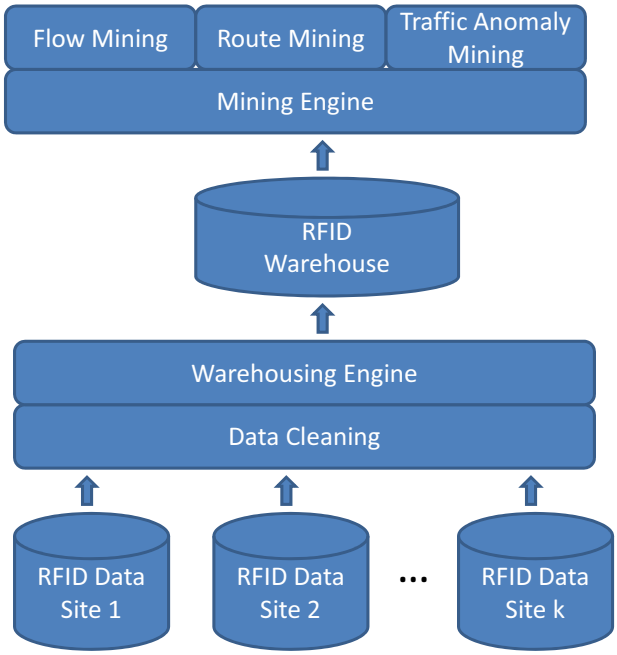
Chapter 1

Introduction

Recent years have witnessed and enormous increase in moving object data from tag readings in supply chain operations, to toll and road sensor readings from vehicles on road networks. A big factor in the emergence of these data, is the rapid adoption of Radio Frequency Identification (RFID) technology in industry and government. RFID, is a technology that allows a sensor (RFID reader) to read, from a distance and without line of sight, a unique identifier that is provided (via a radio signal) by an "inexpensive" tag attached to an item. The technology has applications in many diverse areas. In business, it offers an alternative to bard code identification, that can be used to improve item tracking and inventory management in the supply chain. In traffic engineering, toll booth transponders, can be used to track vehicles in cities, and the information can be used for route recommendations, incident detection, or road upgrade planning. Many other areas, such as tracking of patients in hospitals, monitoring of sensitive assets, and control of medications are currently using RFID to improve their operation.

Large retailers like Walmart, Target, and Albertsons have already begun implementing RFID systems in their warehouses and distribution centers, and are requiring their suppliers to tag products at the pallet and case levels. Cities like San Francisco, are using toll booth readers through the highway system to track vehicles, and aid in traffic control. The main challenge then becomes how can industry and government handle and interpret the enormous volume of data that an RFID application generates. Venture Development Corporation (VDC), a research firm, predicts that when tags are used at the item level, Walmart will generate around 7 terabytes of data every day. Database vendors like Oracle, IBM, Teradata, and some startups are already proving middleware solutions to integrate RFID data into enterprise applications.

Figure 1.1 presents an overview of the data management and mining framework proposed in this dissertation. At the core of the model we have the RFID data warehouse, which stores clean and compressed object movement data in a multi-dimensional model [42]. Feeding data into the warehouse, we have the data cleaning module [45], which receives a stream of raw readings from multiple sources, and executes a cost-conscious cleaning plan to generate clean data. On top of the RFID data warehouse we built a mining engine, which is used to discover high level patterns residing at multiple levels of abstraction. We have implemented three mining modules. The first one, mines commodity flow patterns in the context of supply chain operations [41, 40]. The second and third modules deal with



2

Figure 1.1: Dissertation Overview

route recommendation [43], and traffic anomaly detection [44], in the context of massive road networks.

1.1 RFID Data Warehousing

The increasingly wide adoption of RFID technology by retailers to track containers, pallets, and even individual items as they move through the global supply chain, from factories in producer countries, through transportation ports, and finally to stores in consumer countries, creates enormous datasets containing rich multi-dimensional information on the movement patterns associated with objects and their characteristics. However, this information is usually hidden in terabytes of low-level RFID readings, making it difficult for data analysts to gain insight into the set of interesting patterns influencing the operation and efficiency of the procurement process. In order to realize the full benefits of detailed object tracking information, we need to develop a compact and efficient RFID cube model that provides OLAP-style operators useful to navigate through the movement data at different levels of abstraction of both spatiotemporal and item information dimensions. This is a challenging problem that cannot be efficiently solved by traditional data cube operators, as RFID datasets require the aggregation of high-dimensional graphs representing object movements, not just that of entries in a flat fact table.

We propose to model the RFID data warehouse using a *movement graph*-centric view, which makes the warehouse conceptually clear, better organized, and obtaining significantly deeper compression and an order of magnitude

performance gain over competing models in the processing of path queries. The importance of the *movement graph* approach to RFID data warehousing can be illustrated with an example.

Example. Consider a large retailer with a global supplier and distribution network that spans several countries, and that tracks objects with RFID tags placed at the item level. Such a retailer sells millions of items per day through thousands of stores around the world, and for each such item it records the complete set of movements between locations, starting at factories in producing countries, going through the transportation network, and finally arriving at a particular store where the item it is purchased by a customer. The complete path traversed by each item can be quite long as readers are placed at very specific locations within factories, ships, or stores (*e.g.*, a production lane, a particular truck, or an individual shelf inside a store). Further, for each object movement, there could be many properties, such as the shipping cost, weight loss, or temperature, recorded.

The questions become "*how can we present a clean and well organized picture about RFID objects and their movements?*" and "*whether such a picture may facilitate data compression, data cleaning, query processing, multi-level, multi-dimensional OLAPing, and data mining?*"

Our movement-graph approach provides a nice and clean picture for modeling RFID objects at multiple levels of abstraction. Further, it facilitates data compression, data cleaning, and answering rather sophisticated queries, such as

- High-level aggregate/OLAP query: *What is the average shipping cost of transporting electronic goods from factories in Shanghai, to stores in San Francisco in 2007? and then click to drill-down to month and see the trend*.
- Path query: *Print the transportation paths for the meat products from Argentina sold at L.A. on April 5, that were exposed to over* $40°C$ *heat for over 5 hours on the route.*
- Data mining query: *Why did the 20 packages of Dairyland milk at this Walmart store go bad today? Is it more related to farm, or store, or transportation?* ■

We propose a *movement graph*-based model, which leads to concise and clean modeling of massive RFID datasets and facilitates RFID data compression, query answering, cubing, and data mining. The *movement graph* is a graph that contains a node for every distinct (or more exactly, interesting) location in the system, and edges between locations record the history of shipments (groups of items that travel together) between locations. For each shipment we record a set of interesting measures such as, travel time, transportation cost, or sensor readings like temperature or humidity. We show that this graph can be partitioned and materialized according to its topology to speedup a large number of queries, and that it can be aggregated into cuboids at different abstraction levels according to location, time, and item dimensions, to provide multi-dimensional and multi-level summaries of item movements. The technical contributions can be summarized as follows:

1. **Gateway-based partitioning of the movement graph**. We make the key observation that the *movement graph* can be divided into disjoint partitions, that are connected through special gateway nodes. Most paths with locations in more than one partition include the gateway nodes. For example, most items travel from China to the United States by going through major shipping ports in both countries. Gateways can be given by a user or be discovered by analyzing traffic patterns. Further, materialization can be performed for indirect edges between individual locations and their corresponding gateways and between gateways. Such materialization facilitates computing measures on the paths connecting locations in different partitions. An efficient graph partitioning algorithm, is developed, that uses the gateways to split the graph into clusters in a single scan of the path database.

2. **Redundancy elimination.** RFID readers provide tuples of the form $(EPC, location, time)$ at fixed time intervals. When an item stays at the same location, for a period of time, multiple tuples will be generated. We can group these tuples into a single one of the form $(EPC, location, time\_in, time\_out)$. This form of compression is lossless, and it can significantly reduce the size of the raw RFID readings.

3. **Partition-based bulky movement compression**. Items tend to move and stay together through different locations. For example, a pallet with 500 cases of CDs may arrive at the warehouse; from there cases of 50 CDs may move to the shelf; and from there packs of 5 CDs may move to the checkout counter. We can register a single stay or transition record for thousands of items that stay and move together. Such record would point to a gid, which is a generalized identifier pointing to the subgroups that it contains. In global supply-chain applications, one may observe a "*merge-split*" process, *e.g.*, shipments grow in size as they approach the major ports, and then after a long distance bulky shipping, they gradually split (or even recombine) when approaching stores. We propose a partitioned map table, that creates a separate mapping for each partition, rooted at major shipping ports.

4. **Movement graph aggregation**. The movement graph can be aggregated to different levels of abstraction according to the location concept hierarchy that determines which subset of locations are interesting for analysis, and at which level. For example, the *movement graph* may only have locations inside Massachusetts, and it may aggregate every individual location inside factories, warehouses, and stores to a single node. This aggregation mechanism is very different from the one present in traditional data cubes as nodes and edges in the graph can be merged or collapsed, and the shipments along edges and their measures need to be recomputed using different semantics for the cases of node merging and node collapsing. A second view of *movement graph* aggregation is the process of merging shipment entries according to time and item dimensions.

5. **Partitioned-based cubing**. We propose an efficient cubing algorithm that performs partial materialization of a

partitioned *movement graph* cube. The algorithm first transforms the original path database into a compressed path prefix tree representation, that is used to compute the map table, and perform simultaneous aggregation of paths and items to every interesting level of abstraction.

1.2 RFID Data Cleaning

The success of RFID technology depends heavily on the quality of data it generates, *i.e.*, readers should detect all the tags that are present within their read range, and should not detect tags that are not present, or that are present but due to business rules should not be detected. But the reliability of current RFID systems is far from optimal. Depending on a variety of factors, such as radio frequency (RF) noise, the presence of metal, the generation of the tags, the distance from tags to readers, or even the manufacturer and price of readers, detection rate (the percentage of present tags that are detected) can fluctuate from 100% under ideal circumstances to less than 50% [31, 58, 13, 29] in difficult environments. Data cleaning is thus essential for the correct interpretation and analysis of RFID data, but given the enormous volume of information, diverse sources of error, and rapid response requirements, it can be a challenging task. We will illustrate the difficulties with an example.

Example. A large retailer, with stores throughout the country, may have thousands of RFID readers located at warehouses, distribution centers, and store backrooms. Diversity of factors such as reader location, tag communications protocol, environmental noise, item speed, and item contents among others will be the norm, and will all have an impact on data quality. Such application may require the cleaning of millions of tag readings per second. It is unrealistic to think that a single method will be able to efficiently and accurately clean such enormous volume of data, generated under such diverse set of conditions.

Current approaches to RFID data cleaning [58, 13] have focused on the development of techniques, usually based on smoothing filters, which use the recent history of tag detections at each reader to fill in missed detections. Fixed-window smoothing [13, 29] works well when the detection rate for a tag at a reader is known in advance and is not changing, whereas variable-window smoothing adapts well to changing tag detection rates [58], although at a somewhat higher cost given the need to recompute window sizes for every tag after every reading is received. Using any single cleaning technique without regard to the context in which the reading took place can incur unnecessary costs, when we apply a method more expensive than necessary, *e.g.*, when a reader has consistently good detection rates, and we keep recomputing window sizes, or we apply a method that may be ineffective, *e.g.*, a reader located at a conveyor belt should only detect tags that are moving on the belt and ignore others, but smoothing alone cannot catch such errors. We devise a context-sensitive cleaning method that can choose the most cost-effective cleaning method depending on the conditions in which each reading takes place.

The problem of cost-conscious data cleaning is defined as follows. Given three inputs: (1) a set of tag readings, which form a representative sample of the possible set of readings, each labeled with the correct location for the tag, and also annotated with contextual information, such as item type (metal, water contents), area conditions, or tag protocol; (2) a set of cleaning methods with associated per-tuple cleaning costs; and (3) a per-tuple miss-classification cost, which may be constant, or a function of the tag reading and incorrectly assigned location, our goal is to learn a *cleaning plan* that *identifies the conditions (feature values) under which a specific cleaning method or a sequence of cleaning methods should be applied in order to minimize the expected cleaning costs, including error costs*.

In this dissertation we introduce a framework for the cost-conscious cleaning of RFID data streams, a novel data cleaning approach capable of adapting to the diversity of conditions found in a large RFID implementation, and able to clean large data sets at a minimum cost with high accuracy. We summarize our contribution as follows:

1. **A cost optimization view of RFID data cleaning**: We present a novel formulation of data cleaning as a cost minimization problem. This view allows us to balance accuracy and cleaning costs in order to correctly classify tag readings with minimum resources. This view of the problem is, to the best of our knowledge, the first to address the issues of scalability in cleaning very large volumes of RFID data by taking advantage of the diversity of conditions in which an RFID implementation operates. We identify and categorize the sources of errors, the available cleaning methods, and the features that play an important role in tag detection rates, and that can be used to formulate a good cleaning strategy.

2. **The introduction of the cleaning plan**: In general finding the optimal assignment of cleaning methods to tag reading conditions, such that cost is minimized, is hard. We introduce the cleaning plan, *a decision tree, induced by greedily choosing features that provide the highest* expected cost reduction *as an approximation that works well under a variety of conditions*. Through extensive experiments we show that in general the *cleaning plan* provides higher accuracy at lower cost than any single cleaning method.

3. **The concept of an optimal cleaning sequence**: We introduce the concept of the optimal cleaning sequence as the best possible sequence of cleaning methods to apply to the leaf nodes in the *cleaning plan*. In general finding the *optimal cleaning sequence* is hard and we propose a greedy algorithm that finds a good approximation in practice.

4. **An effective cleaning method based on Dynamic Bayesian Networks (DBNs)**: We propose a new cleaning method that does not require a window to smooth tag detections, but that looks at each reading as a noisy observation of an underlying process that is hidden. This method dynamically adjusts the belief state (probability that the tag is present) given the last observation. Our extensive experiments show that under some conditions DBN-based cleaning can outperform or complement methods based on smoothing windows.

1.3 Flow Mining

Finding flow patterns is a vital aspect of supply chain management. It is important to detect what are the major trends in flow, of different products, by different manufacturers, and distributors in their path from factories to shelves. We propose a method to construct a warehouse of commodity flows [40, 41], called FlowCube. As in standard OLAP, the model will be composed of cuboids that aggregate item flows at a given abstraction level. The FlowCube differs from the traditional data cube in two major ways. First, the measure of each cell will not be a scalar aggregate but a commodity FlowGraph that captures the major movement trends and significant deviations of the items aggregated in the cell. Second, each FlowGraph itself can be viewed at multiple levels by climbing the concept hierarchy of path stages.

There are two forms of aggregation in the FlowCube. **Path View**, where we aggregate the paths traversed by items to different abstraction levels. Path aggregation is done by using the location hierarchy to collapse path stages, *e.g.*, a store manager may collapse all transportation locations into a single node, while expanding other locations. **Item View**, an orthogonal view into RFID commodity flows is related to items themselves, where we aggregate the dimensions describing an item along their corresponding concept hierarchies. This is a view much closer to traditional data cubes.

The measure of each cell in the FlowCube is called a FlowGraph, which is a tree shaped probabilistic workflow, where each node records transition probabilities to other nodes, and the distribution of possible durations at the node. Additionally nodes keep information on exceptions to the general transition and duration distributions given a certain path prefix that has a minimum support (occurs frequently in the data set). We proposed an efficient algorithm to construct the FlowCube based on the following technical principles.

1. **Shared computation.** We explore efficient computation of the FlowCube by conducting the computation of frequent cells and frequent path segments simultaneously. Similar to shared computation of multiple cuboids in BUC-like computation [7], we propose to compute frequent cells in the FlowCube and frequent path segments aggregated at every interesting abstraction level simultaneously.

2. **Pruning of the search space using both the path and item views**. To speed up cube computation, we use pre-counting of high abstraction level itemsets that will help us prune a large portion of the candidate space without having to collect their counts. For example if we detect that the stage shelf is not frequent in general, we know that for no particular duration it can be frequent; or if a store location is not frequent, no individual location within the store can be frequent. Similarly, if the clothing category is not frequent, no particular shirt can be frequent.

3. **Cube compression by removing redundancy and low support counts.** We reduce the size of the FlowCube

by exploring two strategies. The first is to compute only those cells that contain only a minimum number of paths (iceberg condition). This makes sense as the FlowGraph is a probabilistic model that can be used to conduct statistically significant analysis only if there is enough data to support it. The second strategy is to compute only FlowGraphs that are non-redundant given higher abstraction level FlowGraphs.

1.4 Mining Route Recommendations

Efficient fastest path computation in the presence of varying speed conditions on a large scale road network is an essential problem in modern navigation systems. Existing systems compute fastest paths based on road Euclidean distance and a fixed (or in some cases dynamic) model of road speeds. However, "*History is often the best teacher*". Historical traffic data or driving patterns are often more useful than the simple Euclidean distance-based computation because people must have good reasons to choose these routes, *e.g.*, they may want to avoid those that pass through high crime areas at night or that likely encounter accidents, road construction, or traffic jams. Additionally, we also consider factors affecting road speed, such as weather, time of day, and vehicle type, important in selecting fast routes that match the current driving conditions.

Example). Suppose you are new to an area and need to drive to a nearby town to catch a flight at the airport. You would like to get to the airport safely and as quickly as possible. If you have the opportunity to ask a local driver on how to drive there, s/he will likely give you a nice and quick route, although it may not be necessarily the quickest. The suggested route, for example, will not send you through a high crime area, or if there is a snow storm, it will avoid the roads that likely become icy and dangerous. Local experts will consider a multitude of important factors that are difficult to explicitly incorporate into a path finding algorithm. We propose that instead of trying to model all such factors explicitly, we mine historic traffic data and learn from the past driving behavior. In our algorithm we can give preference to fast routes that have high support, *i.e.*, that are frequently traveled, over those, though fast, rarely taken by drivers. ■

In this dissertation, we present a traffic-mining-based path-finding method that first mines speed and driving models from historic traffic data, and when a query is posed to the system (which contains the start point, the end point, and departure time, or some other information, *e.g.*, car pool), it computes the fastest route, based on additional conditions, such as weather forecast, or road construction/closure information. We make the following technical contributions.

1. **Road hierarchy-based partitioning**. We use the natural hierarchy present in road networks to partition the network into semantically meaningful areas. We construct high level areas by dividing the graph using the largest possible roads: Each area at this level is enclosed by large highways and will probably contain a large number of nodes and edges. We recursively subdivide areas by progressively decreasingly the road-scale. An

efficient algorithm is developed that automatically partitions an arbitrary road network and constructs a natural hierarchy of areas. These areas are essential to the algorithm as they will be used to guide the driving pattern mining and adaptive fastest path pre-computation.

2. **Speed rule mining**. We take a traffic database that records observed road speeds under a variety of conditions (*e.g.*, weather, time of day, and accidents) and induce a set of concise rules of the form "if conditions c for edge e then speed factor = f", where speed factor is the speedup or slowdown for an edge with respect to the edge's base-speed. Speed factors are clustered to increase rule support. The concise set of rules is mined through a decision tree induction algorithm.

3. **Driving pattern mining**. We mine frequently traveled edges or edge-sequences in order to obtain important driving hints that are hidden in the data and otherwise difficult to model. We propose a novel area-level mining that computes frequent path-segments at the area level with a support relative to the traffic in the area, *e.g.*, lower support thresholds for edges in rural areas than those in big cities. Such adaptive support will avoid generate too many or too few driving patterns.

4. **Adaptive pre-computation**. Fastest path algorithms usually pre-compute a subset of fastest paths in order to speedup path computation at runtime. The problem is that pre-computation schemes assume unique fastest paths, and when we have variable edge speeds fastest paths be valid only for a given set of conditions. We develop an area-level pre-computation strategy that pre-computes high benefit paths at the area level, *i.e.*, the fastest paths do not change too much. This strategy allows us to speed up query processing without exploding the space requirements.

1.5 Mining Traffic Anomalies

Traffic monitoring is an essential task for transportation management. The volume of vehicles on the road has been increasing at a fast pace, while the number of new roads and freeways has remained relatively stable due to the high economic and social costs of new construction [47]. This trend creates increased pressure on transportation engineers to operate the existing road network as efficiently as possible, guaranteeing a smooth flow of vehicles under high demand conditions, a goal that is difficult to achieve.

Faced with such a challenge, traffic engineers have turned to the development of road monitoring systems, that collect traffic data, such as vehicle flow, and speed, for large portions of the road network. In 2000, the federal government started an initiative to establish a national level transportation data network (*e.g.* Federal Highway Administration's Mobility Monitoring Program); several states (*e.g.*, California, Washington State, Virginia) have also

developed transportation data archive systems to integrate operational transportation data with road system geometrics and characteristics. Such historic data are necessary for proactive traffic operations management in order to forecast the likelihood of problematic sections and assist in the preparation of strategies to mitigate anticipated problems. A collective data pool increases the cost effectiveness of data retention and storage activities for performance measure development, transportation planning and design, and traffic management and operations. The emergence of enormous archived transportation data for proactive traffic management imposes a pressing need for advanced data mining algorithms for efficient data processing and knowledge extraction.

One important application of traffic monitoring is anomaly detection. A typical anomaly can be a large drop in speed in a section of highway, or an abnormal increase in the flow of vehicles to an area of the city. A significant effort has been made on incident detection, which tries to automatically identify incidents given abnormal changes in traffic. Algorithms for this purpose have had very limited success in practice, and due to large numbers of false positives their alerts are largely ignored. Rather than focusing on incident detection this work considers the mining and exploration of anomalies in multi-dimensional space. By building a framework that allows traffic engineers to overlay and cluster anomalies according to multiple criteria (*e.g.*, topology or severity) and along different contextual dimension (*e.g.*, weather conditions, road network area, or time of day), we provide a powerful analysis tool, that through OLAP-style operations, can be used to discover interesting patterns associated with traffic anomalies. Such patterns shed light, not only on factors related to incidents, but also into a broader class of traffic problems, including road network design, traffic prediction, and route planning.

Example (Atypical Fragment). California monitors its road network with more than 10,000 sensors located in freeways throughout the state. This system generates in excess of 1 gigabyte of sensor data per day. If we include data from cameras and RFID sensors, the volume is orders of magnitude larger. These traffic data sets contain thousands of abnormal observations per minute, which can overwhelm both human operators and automated algorithms trying to find incidents in the road network. We can significantly reduce the number of individual anomalies, by aggregating events that occur in close spatial and temporal proximity into a single anomaly that is in turn described by semantically rich properties, such as diameter, duration, or propagation speed (features not available at the individual anomaly level), and that can be more easily understood and interpreted. Such reduction in the number of detected anomalies, can improve the performance of a multitude of algorithms, such as incident detection, anomaly overlay, and anomaly cubing, which are highly sensitive to the size of their input ■

Example (Anomaly Overlay). A typical road monitoring system generates hundreds or even thousands of anomalies every day. Such anomalies are associated with contextual conditions (*e.g.*, weather, time of day, and geographic location) and can be described by properties such as topology, severity, or other spatiotemporal characteristics. It would be helpful to navigate the very large space of anomalies, according to their properties and context. For example,

one may overlay all the anomalies of similar severity at the city level, mine discriminating characteristics associated with high and low severity clusters, and then drill down to specific areas of the road network and explore the effects of weather on anomaly severity. Alternatively, one may overlay anomalies according to road topology and correlate severity with road configurations such as interchanges and highway merges. Such analysis requires a method to overlay and cluster anomalies according to a multitude of interesting factors. The development of an anomaly cube may facilitate efficient navigation through different anomaly overlays in multi-dimensional space. ■

In this dissertation, we propose a comprehensive framework for discovery, overlay and cubing of traffic anomalies from massive traffic data, with the following main contributions.

1. **Atypical fragment mining**. An *atypical fragment* is a subset of connected roads that have significant changes in speed during overlapping time intervals. We introduce atypical fragment as a compact, yet expressive representation of traffic anomalies, which provides a concise picture of an anomaly, aggregating a good number of related anomalous events into a single event, in a way similar to how a human would identify a congestion in a large segment of highway as a single incident (not as thousands of independent anomalies), spanning miles of highway, and with a duration of many minutes.

   An efficient algorithm, AFragMine, is developed to identify the set of atypical fragments in a large data set of traffic readings generated by road sensors. First, a model of traffic behavior under different conditions is developed and used to derive single road segments that present anomalous traffic characteristics. Then AFragMine computes all sequences of edges spatially and temporally connected through paths of change in a single scan of the traffic database.

2. **Atypical fragment overlay**. We introduce a model to overlay atypical fragments (anomalies) according to a wide range of criteria, and present three important types of overlay: (1) *topology-based*, which merges atypical fragments that share the general topology at the road level; (2) *severity-based*, which overlays atypical fragments that share similar severity characteristics (*e.g.*, similar total duration, and length); and (3) *spatiotemporal-based*, which overlays atypical fragments that occur in close spatial and temporal proximity over a period of time. Note that the first overlay allows the semantic merging of atypical fragments that share similar basic structures at the road level; the second overlay is important at identifying recurrent patterns associated with anomalies in different severity layers; finally the third can be used to identify, recurrent bottlenecks, which are a critical concept important for incident detection and traffic prediction.

   Our proposed framework places the atypical fragment overlay, in a multi-dimensional cubing model, that allows for OLAP-style navigation of anomaly clusters. Each cell in the cube contains a set of contextual dimensions describing a group of anomalies that share the same dimension values, and the cube measure is an overlay of

the anomalies in the cells. Such model facilitates user-guided exploration of anomalies at multiple levels of abstraction of the contextual dimensions. We show that several optimization techniques are possible in order to materialize cells of the cube efficiently. The anomaly cube is a novel model, both in traffic engineering and in data mining, and can be use as the basis to tackle a variety of new mining problems in the area of traffic management.

We implemented our proposed framework, and populated it with data from the California road network. We developed modules, for data cleaning, atypical fragment mining, and multi-dimensional overlay. We show in the experimental section, through qualitative and quantitative results, that the framework is a powerful tool to efficiently mine and analyze large traffic data sets.

1.6 Organization

The rest of the dissertation is organized as follows. Chapter 2, presents a review of literature relevant to moving object data, and RFID related technologies. Chapter 3, gives a short introduction to RFID technology, and data generation processes. Chapter 4, proposes compression and cubing techniques to warehouse RFID data. Chapter 5, introduces a cost-conscious cleaning method that can clean large volumes of raw RFID readings. Chapter 6, defines the FlowGraph and presents an efficient computation algorithm. Chapter 7, extends the FlowGraph into a multi-dimensional data cube model called the FlowCube. Chapter 8, presents the problem of route recommendations on a large road network. Chapter 9, proposes a mining and cubing model for traffic anomalies. We conclude the dissertation and propose future research directions, in chapter 10.

Chapter 2

Literature Review

2.1 Online Management of RFID Data

Significant effort has been spent in the development of an scalable and robust architecture for the online management of RFID data. This line of work presents a global architecture for management of tag readings, called the EPC Global Network. The framework provides a standard useful for the communication of different components, but does not define the detailed operation of those. For example, it mentions components for cleaning, warehousing, and mining as black boxes that make part of the network.

The EPC Global Network [81] is composed of the following components. Savants [82], which are computers that receive a stream of tag readings from RFID readers, and perform filtering of the streams. The readers scan for tags at periodic intervals, and provide an Electronic Product Code (EPC) [12] that uniquely identifies each item in its detection range. Typical functions of the Savant filters, are to clean and smooth the data, remove uninteresting readings, and construct higher level events from low level ones. Savants can be stacked in a hierarchy, with lower level ones operating on raw EPC detection events, while higher level ones operating on preprocessed events. The number, and functionality of the savants will be dependent on the application, and is not part of the specification. Consuming events from the Savants, we have the EPC Information Service (EPCIS), which are applications that communicate with systems such as point of sale, and inventory management. EPICS, uses the Object Naming Service (ONS) [69] to find the service, usually an URL, that provides detailed information on a particular EPC. The ONS provides data in a language known as Physical Markup Language (PML) [30], this data can be static, such as the manufacturer and expiration date of the product, or dynamic, such as the set of locations it has visited. Each company can have its own EPC Network, with all the networks communicating among them through the Internet.

Our work on RFID data management, fits nicely on the EPC Global Network framework. Our cost-conscious cleaning method, can be implemented at the savant level, while our warehousing and mining models would the components in the EPC Global Network that communicate with the EPCIS layer to receive a stream of tag detection events, at multiple company sites, and will integrate the information into a centralize repository.

2.2 RFID Data Warehousing

A data warehouse is an enterprise level data repository that collects and integrates organizational data in order to provide decision support analysis. At the core of the data warehouse is the data cube, which computes an aggregate measure (*e.g.*, sum, avg, count) for all possible combination of dimensions of a fact table (*e.g.*, sales for 2004, in the northeast). Online analytical processing (OLAP) operations provide the means for exploration and analysis of the data cube.

An RFID data warehouse shares many common principles with the traditional data cube [1, 17, 46]. They both aggregate data at different levels of abstraction in multi-dimensional space. Since each dimension has an associated concept hierarchy, both can be (at least partially) modeled by a *Star* schema. The problem of deciding which cuboids to construct in order to provide efficient answers to a variety of queries specified at different abstraction levels is analogous to the problem of partial data cube materialization studied in [52, 84]. However, cubing RFID data, differs from a traditional data cube in that it also models *object transitions* in multi-dimensional space, which is crucial to answer queries related to the trajectories taken by objects in the system, both at the raw level, and more interestingly, at high abstraction levels, *e.g.*, how many items went from factories in China to retail stores in the Northeast by means of the port in New York. Trajectory aggregation would require a traditional data cube to have an unmanageable number of dimensions (*i.e.*, one dimension per distinct combination of product, location, time_in, time_out, and parent), which is infeasible in traditional models.

Efficient computation of data cubes, has been a major concern for the research community. Computing a full cube is a challenging problem, that is exponential in the number of dimensions of the fact table. Multi-way array aggregation [94], is a powerful method for full cube computation. It uses the idea of shared computation, and simultaneous aggregation to reduce the number of scans of the fact table, and to compute high level measures from lower level ones without having to read the original database. Iceberg cubes, which compute the subset of cells of the full cube which pass a minimum support threshold, are another popular method to speedup computation. Methods on this front include, BUC [7], which conducts iceberg pruning from low dimensional to high dimensional cuboids, and Star Cubing [92], which combines pruning and shared computation to improve efficiency. Our algorithms for computing the RFID data cube, and the cube of RFID workflows adapts techniques related to traditional data cube computation, to the aggregation of trajectories, and workflows in the context of RFID data sets. Partial materialization of data cubes, is another technique to reduce the computation complexity, the idea is to materialize only a subset of the cuboids, and use those to compute any cuboid on the fly. Partial materialization techniques such as [52], try to select the subset of cuboids, that for a given query load, provide the best benefit per unit of storage. Another method, is presented in [50], which pre-computes popularly request cuboids, and cuboids along frequent paths between those. Given the very high dimensionality of RFID data, we make use of partial materialization in all of our cubing algorithms.

2.3 RFID Data Cleaning

Cleaning of RFID data has been addressed both from the design of robust communication protocols, that operate at the hardware level, and from post-processing software designed to detect incorrect readings. There have been extensive studies in the design of communication protocols, at the hardware level, which are robust to interference and collisions [55, 39]. At the software level, the use of smoothing windows has been proposed as a method to reduce false negatives. [31] proposes the fixed-window smoothing. [58] proposes a variable-window smoothing that adapts window size dynamically according to tag detection rates, by using a binomial model of tag readings. [59] presents a framework to clean RFID data streams by applying a series of filters, but does not go into the details of filter design. Our work differs from these studies in that our cost-conscious cleaning framework operates at a higher level than any single technique and learns how to apply multiple techniques to increase accuracy and reduce costs. We also introduce a new technique based on Dynamic Bayesian Networks [79] that smooths data by updating a belief state rather than by recording the history of recent readings.

Our work takes advantage of previous studies on management and mining of RFID data sets [42, 40] to incorporate important contextual information into the cleaning process and take it as a source of labeled data. We use the Flow-Graphs proposed in [40] to develop a more sophisticated *DBN* that incorporates historic transition likelihoods and group movement characteristics into the transition and observation models of the *DBN*. These repositories of RFID data can also be used to obtain labeled information necessary for the induction of the cleaning plan.

A concept close to our cost-conscious cleaning is cost-conscious classification. This idea has been extensively studied in the AI literature. [88] compares a number of cost-conscious decision tree induction algorithms. These approaches deal with the problem of constructing a decision tree that favors the use of inexpensive features over more expensive ones. The criterion for selecting features at each node in the tree is a variation of information gain that incorporates feature cost. These methods focus on a single classification technique, namely the decision tree, and try to construct a tree that can be used inexpensively. We instead focus on how to apply multiple cleaning methods (classifiers) at minimum cost and with high accuracy.

2.4 Mining Flow Trends

Mining flow trends from RFID data sets, shares many characteristics with the problem of process mining [89]. Workflow induction, studies the problem of discovering the structure of a workflow from event logs represented by a list of tuples of the form $(case_i, activity_j)$ sorted by the time of occurrence; $case_i$ refers to the instance of the process, and $activity_j$ to the activity executed. [2] first introduced the problem of process mining and proposed a method to discover workflow structure, but for the most part their methods assumes no duplicate activities in the workflow, and

does not take activity duration into account, which is a very important aspect of RFID data, where an object may visit the same location multiple times, and the duration that it spends at each location is vital in understanding its flow properties.

Another area of research very close to flow analysis is is that of grammar induction [15, 87], the idea is to take as input a set of strings and infer the probabilistic deterministic finite state automaton (PDFA) that generated the strings. This approach is inadequate for RFID flow analysis, because the induced grammar does not distinguish major flow trends, from flow exceptions, and if we define each combination of time and location as a symbol of the alphabet the size of the grammar would be enormous and the volume of training data, required to learn a reliable model would be prohibitive.

2.5 Mining Route Recommendations

Shortest path computation is an area that has received extensive research attention for almost half a century. There are so many results on this area that we can only highlight a few selected publications to put our work into perspective. A good survey of shortest path methods is found in [70, 35].

Exact algorithms for static graphs have used the idea of edge hierarchies to reduce computation time. [62, 61] and more recently [80, 23] have looked at the techniques to decompose the graph using hierarchies and pre-compute selected paths. The hierarchy defined by these methods is similar in spirit to ours, but theirs is usually graph theoretic and not based on the physical size of roads. Approximate algorithms have also considered the idea of graph partitioning. [20] uses large roads to partition the graph, but they do it manually, and no driving or speed patterns are considered. Other graph-theoretic algorithms have focused on improving the heuristics used by $A*$, [24] is a recent example of work along this direction. It uses the concept of landmarks to improve route cost estimation. These algorithms are a complement to our method, we could directly use them to estimate $h(n)$ and improve performance.

Shortest path computation on dynamic graphs has two interpretations. The first is graphs where edges are updated, deleted, and added. Under this interpretation the idea of most methods is to apply a static shortest path algorithm such as Dijsktra's [25] only to the subset of the graph where fastest paths change after the upgrade. [34, 24] compare the performance of a few of the most popular single source dynamic shortest path methods. [66] presents an all pairs dynamic fastest path algorithm. The second interpretation is that edge speed is a function of time. [63] looks at this problem, they define the CapeCod pattern to match time of day to edge speed. Their algorithm is an adaptation of A*: Instead of sorting the priority queue on a scalar cost, they maintain a function of cost based on starting time. The focus of this work is to provide the complete set of fastest paths for a given time interval. This technique can be used by our algorithm in performing path pre-computation at the area level. [6] uses a statistical approach based on clustering to

compute rules useful in predicting fastest paths. We are different to this work in that our approach to route finding is search, not prediction.

2.6 Mining Traffic Anomalies

[37] gives a good introduction to the basic concepts of transportation engineering. Inductive loop detectors are studied in [64], and the reliability problems empirically verified in [8]. Data cleaning techniques for loop detector data are proposed in [18], and effective methods for speed estimation on single loop detectors are introduced in [90].

Incident detection in a road network is a problem that has received significant attention for the past 30 years. The main goal of incident detection has been the automatic identification of important events such as accidents or breakdowns, but the goal has proved elusive. In our work we have focused on the problem from a very different angle. We do not aim to identify incidents, but to provide a framework that can be used by traffic engineers to analyze anomalies efficiently and in multi-dimensional space. Works on incident detection include, early algorithms, which focused on developing incident detection rules based on occupancy [73, 22]. More recent work has used neural networks [60], Fuzzy Logic [16], Support Vector Machines [19], and adaptive clustering [9] to attack the problem. For an excellent survey on the topic see [48, 75]. Patterns discovered by analysis through the analysis of anomaly data in our propose framework can be used as input to incident detection algorithms, or to a multitude of other traffic engineering problems such as traffic bottleneck forecasting [57].

In the development of our methods we make use of different data mining techniques. Clustering [54] is used to overlay atypical fragments, data cubing techniques such as [7] can be used for efficient anomaly cube computation. [27] studies the problem of incremental clustering, but their problem setup is to refine a pre-computed clustering, with the addition of a few new data points, whereas our shared computation model is to merge multiple existing clusterings.

2.7 Moving Object Research and RFID

RFID data sets are a form of moving object data. The main difference with traditional moving object research, is that the former assumes free movements, *e.g.*, animal migration patterns, or storm trajectories, while the latter only records movements at fixed locations, namely where RFID readers have been installed. For example, items moving in the supply chain and being detected at predetermined locations in factories, warehouses, and stores. In RFID systems, the particular path taken between two reader locations, is usually not important. For example, when a warehouse manager receives a shipment of items from a factory, he does not care about the particular route taken by truck to arrive at the store. A similar case is observed with vehicles moving on a road network, the location of vehicle detection stations (VDS) is well known and fixed, and the path taken by a vehicle between two VDSs is either fixed, or unimportant,

given that the movement of vehicles is constrained by the structure of the road network.

The idea of spatial data cubes was introduced in [50, 86], their proposal considers spatial, and non-spatial dimensions in the data cube, with regions in space, and scalar values as measures. In their work they consider arbitrary spatial objects and aggregate them according to spatial hierarchies. This is very different to path aggregation in an RFID environment, where locations are well defined, and have a clear concept hierarchy; aggregation in RFID does not depend on the geometric characteristics of locations. [71], is a more recent work along cubing spatial databases. Their ideas focus on the development of indexing techniques that can be used to create spatial hierarchies, and that improve processing of OLAP queries.

Frequent trajectory mining, and co-location pattern mining, is another area related to RFID flow analysis. The idea of co-location mining is to find patterns where items match spatial constraints such as being near to each other. Several studies [67, 83, 93] have modified association rule mining algorithms to incorporate spatio-temporal constraints. The problem with applying this line of work to RFID flow analysis, is that locations that are very far apart, but that exhibit correlations in transition probabilities, or duration probabilities should still be discovered, even though they violate co-location constraints. Moving object clustering [65, 36, 14], is also related to the discovery of flow trends, *i.e.*, trajectories that belong to the same cluster may all support the same trend. But these studies, focus on trajectories that can go through any point in space, and for which there is no well defined concept hierarchy. In the case of RFID trajectories are limited to fixed points, and very different trajectories (in spatial terms) can support the same flow trend, when their locations are aggregated to higher abstraction levels.

Chapter 3

An Introduction to RFID Technology

3.1 RFID Concepts

An RFID system is composed of readers (interrogators) and tags which are attached to items. Readers communicate with tags, from a distance, and without line of sight, and interrogate the tags within range for their EPC (Electronic Product Code). Communication between tags and readers is done via RF (Radio Frequency) signals, and it follows standard protocols designed to minimize interference [55, 39]. The communication protocols provide the following functionality.

Tag Inventory: This is the process by which the reader sends an RF signal and receives responses from tags within range. Tag-tag collisions are avoided with a *singulation* mechanism that directs tags to respond one by one. The process of inventorying all tags is also called an *interrogation cycle*.

Tag Selection: This mechanism allows the reader communicate with a subset of the tags. In general, it is done through a selection mask: Only tags whose EPCs match the flag respond to commands.

Tag Access: This allows the reader to read, write (for writable tags), or kill (permanently disable) a single tag.

Figure 3.1 presents the main components of an RFID application. The RFID tag, on the left is attached to the item that is being tracked. The tag is composed of two components, an antenna, which is used to receive and transmit signals from and to the reader, and an integrated circuit that stores information such as the EPC. The reader, interrogates all present tags for their EPCs, and transmits each tag detection to a server that collects readings, does some initial cleaning, and assembles higher level events that are consumed by different applications.

3.2 RFID Data

The operation of an RFID system generates a stream of data resulting from *interrogation cycles* occurring at recurring time intervals at each reader. The data generated for each *interrogation cycle* is usually a set of tuples of the form $(EPC, Reader, time)$. The tuple can be augmented with extra information, such as the $antenna$ used by the Reader, the $power\ level$ of the interrogation signal, or the $tag\ type$ (class 0, class 1, generation 2, *etc.*). From the application

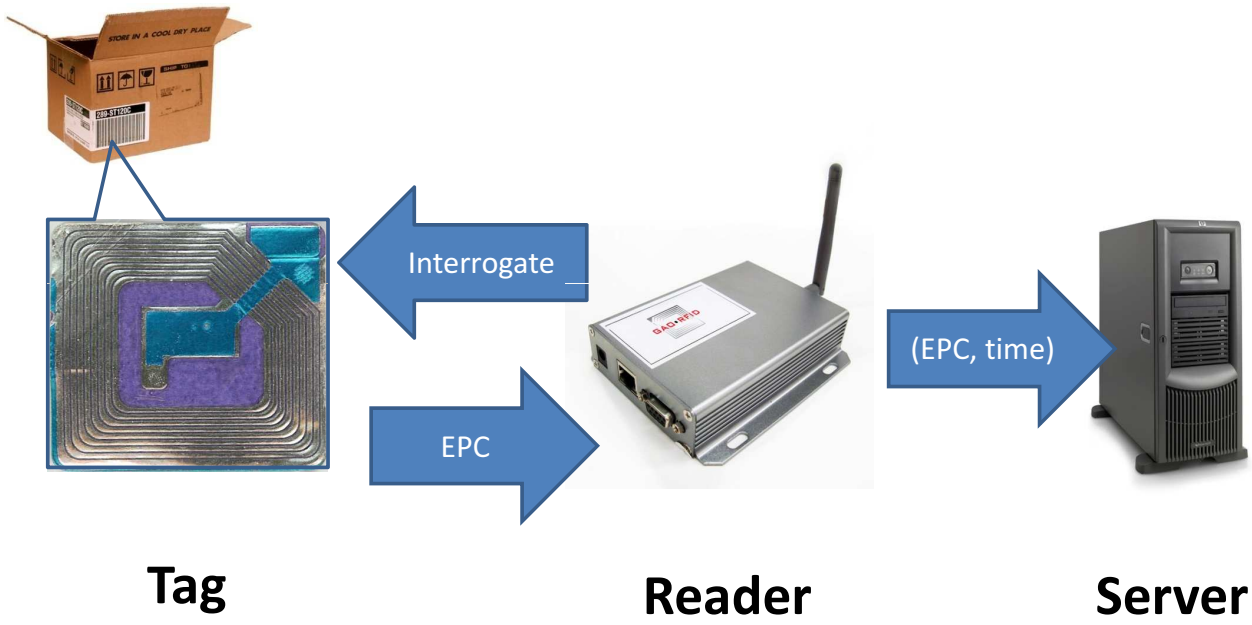


Figure 3.1: Data collection

perspective it is possible to look at multiple *interrogation cycles* as a single unit. The Application Level Event (ALE) specification [5] calls this unit a *read cycle*. When the *read cycle* contains more than one *interrogation cycle*, we may receive a single tuple per *read cycle* with the form: $(EPC, Reader, time, responses)$, where $responses$ is the number of *interrogation cycles* when the tag identified by EPC was correctly inventoried.

Table 3.1 is an example of a raw RFID database where a symbol starting with r represents an RFID tag, l a location (reader), and t a time. The total number of records in this example is 188.

Raw Stay Records
$(r1, l1, t1)$ $(r2, l1, t1)$ $(r3, l1, t1)$ $(r4, l1, t1)$ $(r5, l1, t1)$
$(r6, l1, t1)$ $(r7, l1, t1)$ ... $(r1, l1, t9)$ $(r2, l1, t9)$ $(r3, l1, t9)$
$(r4, l1, t9)$ ... $(r1, l1, t10)$ $(r2, l1, t10)$ $(r3, l1, t10)$ $(r4, l1, t10)$
$(r7, l4, t10)$ ... $(r7, l4, t19)$ ... $(r1, l3, t21)$ $(r2, l3, t21)$
$(r4, l3, t21)$ $(r5, l3, t21)$ ... $(r6, l6, t35)$ ... $(r2, l5, t40)$
$(r3, l5, t40)$ $(r6, l6, t40)$ ... $(r2, l5, t60)$ $(r3, l5, t60)$

Table 3.1: Raw RFID records

In order to reduce the large amount of redundancy in the raw data, data cleaning should be performed. The output after data cleaning is a set of clean stay records of the form $(EPC\_code, location, time\_in, time\_out)$ where *time_in* is the time when the object enters the location, and *time_out* is the time when the object leaves the location.

Data cleaning of stay records can be accomplished by sorting the raw data on EPC code and time, and generating *time_in* and *time_out* for each location by merging consecutive records for the same object staying at the same location.

Table 3.2 presents the RFID database of Table 3.1 after cleaning. It has been reduced from 188 records to just 17 records.

EPC_code	$Stay(EPC\_code, location, time\_in, time\_out)$
r1	$(r1, l1, t1, t10)(r1, l3, t20, t30)$
r2	$(r2, l1, t1, t10)(r2, l3, t20, t30)(r2, l5, t40, t60)$
r3	$(r3, l1, t1, t10)(r3, l3, t20, t30)(r3, l5, t40, t60)$
r4	$(r4, 1, t1, t10)$
r5	$(r5, l2, t1, t8)(r5, l3, t20, t30)(r5, l5, t40, t60)$
r6	$(r6, l2, t1, t8)(r6, l3, t20, t30)(r6, l6, t35, t50)$
r7	$(r7, l2, t1, t8)(r7, l4, t10, t20)$

Table 3.2: A cleansed RFID database

Chapter 4

RFID Data Warehousing

A data warehouse is an enterprise level data repository that collects and integrates organizational data in order to provide decision support analysis. At the core of the data warehouse is the data cube, which computes an aggregate measure (*e.g.*, sum, avg, count) for all possible combination of dimensions of a fact table (*e.g.*, sales for 2004, in the northeast). Online analytical processing (OLAP) operations provide the means for exploration and analysis of the data cube. My research on this direction has extended the data cube to handle moving object data sets [42], by significantly compressing such data, and proposing a new aggregation mechanism that preserves its path structure. The RFID warehouse is built around the concept of the *movement graph*, which records both spatio-temporal and item level information in a compact model. We show that compression and query processing efficiency can be significantly improved, by partitioning the *movement graph* around gateway nodes, which are special locations connecting different spatial regions in the graph.

Before we describe our proposed architecture for warehousing RFID data, it is important to describe why a traditional data cube model would fail on such data. Suppose we view the cleansed RFID data as the fact table with dimensions $(EPC, location, time\_in, time\_out : measure)$. The data cube will compute all possible group-bys on this fact table by aggregating records that share the same values (or any *) at all possible combinations of dimensions. If we use count as measure, we can get for example the number of items that stayed at a given location for a given month. The problem with this form of aggregation is that it does not consider links between the records. For example, if we want to get the number of items of type "dairy product" that traveled from the distribution center in Chicago to stores in Urbana, we cannot get this information. We have the count of "dairy products" for each location but we do not know how many of those items went from the first location to the second. We need a more powerful model capable of aggregating data while preserving its path-like structure.

We propose an RFID warehouse architecture that contains a fact table, *edge*, composed of cleansed RFID records, registering transitions between locations[1]; an information table, *info*, that stores path-independent information for each item, i.e., SKU information that is constant regardless of the location of the item such as manufacturer, lot number, color, etc.; and a *map* table that links together different records in the fact table that form a path. Figure 4.1 shows a

[1] alternatively we can also use a stay table registering stage information

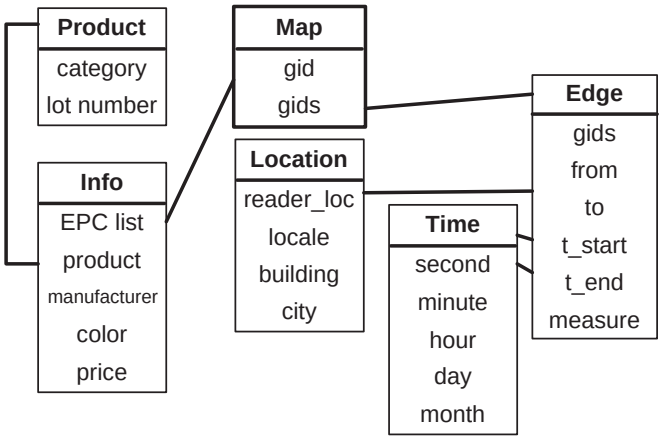


Figure 4.1: RFID warehouse - logical schema

logical view into the RFID warehouse schema. We call the *edge*, *transition*, *info*, and *map* tables aggregated at a given abstraction level an RFID-Cuboid.

The main difference between the RFID warehouse and a traditional warehouse is the presence of the map table linking records from the fact table in order to preserve the original structure of the data.

The computation of RFID-cuboids is more complex than that of regular cuboids as we will need to aggregate the data while preserving the structure of the paths at different abstraction levels.

4.1 Gateway-Based Movement Graph

Among many possible models for RFID data warehouses, we believe the gateway movement graph model not only provides a concise and clear view over the movement data, but also facilitates data compression, querying, and analysis of massive RFID datasets (which will be clear later).

Definition 4.1.1 *A movement graph $G(V, E)$ is a directed graph representing object movements; V is the set of locations, E is the set of transitions between locations. An edge $e(i, j)$ indicates that objects moved from location v_i to location v_j. Each edge is annotated with the history of object movements along the edge, each entry in the history is a tuple of the form $(t_{start}, t_{end}, tag\_list : measure\_list)$, where all the objects in tag_list took the transition together, starting at time t_{start} and ending at time t_{end}, and $measure\_list$ records properties of the shipment.*

Figure 4.2 presents the *movement graph* for the path database in Table 4.1.

A large RFID dataset may involve many locations and movements. It is important to put such movements in an organized picture. In a global supply chain it is possible to identify important locations, that serve to connect remote sets of locations in the transportation network. Gateways generally aggregate relatively small shipments from many

Tag	Path
t1	$(A, 1, 2)(D, 4, 5)(G_1, 6, 8)(G_2, 9, 10)(F, 11, 12)(J, 14, 17)$
t2	$(A, 2, 3)(D, 4, 5)(G_1, 6, 8)(G_2, 9, 10)(F, 11, 12)(K, 16, 18)$
t3	$(A, 2, 3)(D, 4, 6)$
t4	$(B, 1, 2)(D, 3, 5)(G_1, 6, 7)(G_2, 9, 10)(I, 11, 13)(K, 14, 15)$
t5	$(C, 1, 3)(E, 4, 5)(G_1, 6, 7)(G_2, 9, 10)(I, 11, 14)$
t6	$(A, 3, 3)(B, 4, 5)(I, 10, 11)$

Table 4.1: An example path database

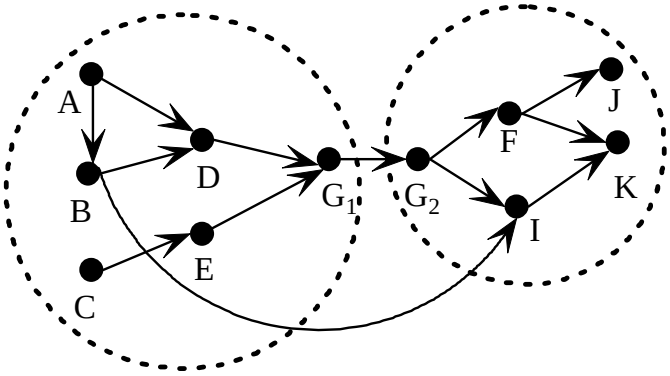


Figure 4.2: An example movement graph

distinct, regional locations into large shipments destined for a few well known remote locations; or they distribute large shipments from remote locations into smaller shipments destined to local regional locations. These special nodes are usually associated with shipping ports, *e.g.*, the port in Shanghai aggregates traffic from multiple factories, and makes large shipments to ports in the United States, which in turn split the shipments into smaller units destined for individual stores. The concept of gateways is important because it allows us to naturally partition the *movement graph* to improve query processing efficiency and reduce the cost of cube computation.

Gateways can be categorized into three classes: *Out-Gateways, In-Gateways*, and In-Out-Gateways, as shown below.

4.1.1 Out-Gateways

In the supply chain it is common to observe locations, such as ports, that receive relatively low volume shipments from a multitude of locations and send large volume shipments to a few remote locations. For example, a port in Shanghai may receive products from a multitude of factories and logistics centers throughout China to later send the products through ship to a port in San Francisco. We call this type of node an *Out-Gateway*, and it is characterized by having multiple incoming edges with relatively low average shipment sizes and a few outgoing edges with much larger average shipment sizes and traffic. One important characteristic of these locations is that products

usually can only reach remote locations in the graph by first going through an *Out-Gateway*. Figure 4.3-a presents an *Out-gateway*. For our running example, Figure 4.2, location G_1 is an *Out-gateway*.

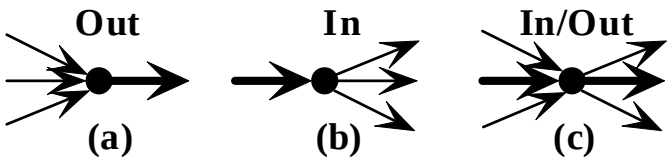


Figure 4.3: Three types of gateways

4.1.2 In-Gateways

In-gateways are the symmetric complement of *Out-Gateways*, they are characterized by a few incoming edges with very large average shipment sizes and traffic, and a multitude of relatively low average shipment size outgoing edges. An example of an *In-Gateway* may be sea port in New York where a large volume of imported goods arrive at the United States and are redirected to a multitude of distribution centers throughout the country before reaching individual stores. As with *Out-Gateways* these nodes dominate a large portion of the flow in the *movement graph*; most products entering a partition of the graph will do so through an *In-Gateway*. Figure 4.3-b presents an example *Out-gateway*. For our running example, Figure 4.2, location G_2 is an *In-gateway*.

4.1.3 In-Out-Gateways

In-Out-gateways are the locations that serve as both *In-gateways* and *Out-gateways*. This is the case of many ports that may for example serve as an *In-gateway* for raw materials being imported, and an *Out-gateway* for manufactured goods being exported. Figure 4.3-c presents such an example. It is possible to split an *In-Out-gateway* into separate In- and Out- gateways by matching incoming and outgoing edges carrying the same subset of items into the corresponding single direction traffic gateways.

Notice that gateways may naturally form hierarchies. For example, one may see a hierarchy of gateways, *e.g.*, country level sea ports →region level distribution centers →state level hubs.

4.2 Data Compression

4.2.1 Redundancy Elimination

RFID data contains large amounts of redundancy. Each reader scans for items at periodic intervals, and thus generates hundreds or even thousands of duplicate readings for items in its range, that are not moving. For example, if a pallet

stays at a warehouse for 7 days, and the reader scans for items every 30 seconds, there will be 20,160 readings of the form (EPC, $warehouse$, $time$). We could compress all these readings, without loss of information, to a single tuple of the form (EPC, $warehouse$, $time\_in$, $time\_out$), where $time\_in$ is the first time that the EPC was detected in the warehouse, and $time\_out$ the last one.

Redundancy elimination can be accomplished by sorting the raw data on EPC and time, and generating *time_in* and *time_out* for each location by merging consecutive records for the same object staying at the same location.

4.2.2 Bulky Movement Compression

Since a large number of items travel and stay together through several stages, it is important to represent such a collective movement by a single record no matter how many items were originally collected. As an example, if 1,000 boxes of milk stayed in location loc_A between time t_1 (time_in) and t_2 (time_out), it would be advantageous if only one record is registered in the database rather than 1,000 individual RFID records. The record would have the form: $(gid, prod, loc_A, t_1, t_2, 1000)$, where 1,000 is the count, *prod* is the product id, and gid is a generalized id which will not point to the 1,000 original EPCs but instead point to the set of new gids which the current set of objects move to. For example, if this current set of objects were split into 10 partitions, each moving to one distinct location, gid will point to 10 distinct new gids, each representing a record. The process iterates until the end of the object movement where the concrete EPCs will be registered. By doing so, no information is lost but the number of records to store such information is substantially reduced.

The process of selecting the most efficient grouping for items, both in terms of compression and query processing, depends on the movement graph topology.

Split-Only Model

In some applications, the movement graph presents a tree-like structure, with a few factories near the root, warehouses and distribution centers in the middle, and a large number of individual stores at the leaves. In such topology, it is common to observe items moving in large groups near the factories, and splitting into smaller groups as they approach individual stores. We say that movement graphs with this topology present an *Split-Only* model of object movements.

In the *Split-Only* model, we can gain significant compression by creating a hierarchy of $gids$, rooted at factories where items move in the largest possible groups, and pointing to successively smaller groups as items move down the supply chain. In this model a single grouping schema provides good compression, because the basic groups, in which objects move, are preserved throughout the different locations, *i.e.*, the smallest groups that reach the stores are never shuffled, but are preserved all the way from the Factory. In the next section we will present a more general model that can accommodate both split and merging of groups.

Merge-Split Model

A more complex model of object movements is observed in a global supply chain operation, where items may merge, split, and groups of items can be shuffled several times. One such case is when items move between exporting and importing countries. At the exporting country, items merge into successively large groups in their way from factories to logistic centers, and finally to large shipping ports. In the importing country, the process is usually reversed, items split into successively smaller groups as they move from the incoming port, to distribution centers, and all the way to individual stores. We say that movement graphs with this topology present an *Merge-Split* model of object movements.

A single object grouping model, such as the one used in a *Split-Only* model would not be optimal when groups of items can both split and merge. A better option is to partition the movement graph around gateways, and define an item grouping model at the partition level. For example, the exporting country would get a hierarchy of groups rooted at the port and ending at the factories, while the importing country will have a separate hierarchy rooted at the port and ending at the individual stores. Using a single grouping for both partitions has the problem that each group would have to point to many small subgroups, or even just individual items, that are preserved throughout the entire supply chain, after multiple operations of merge, split, and shuffle. Separate groupings prevents this problem by requiring bulky movement only at the partition level, and allowing for merge, split, and even shuffling of items without loss of compression.

4.2.3 Data Generalization

Since many users are only interested in data at a relatively high abstraction level, data compression can be explored to group, merge, and compress data records. This type of compression as opposed to the previous two compression methods is lossy, because once we aggregate the data at a high level of abstraction, *e.g.*, time aggregated from second to hour, we can not ask queries for any level below the aggregated one. We identify two types of data generalization, the first is on path dimensions, and the second is on item dimensions.

Path Level Generalization

A new type of data generalization, not present in traditional data cubes, is that of merging and collapsing path stages according to time and location concept hierarchies. For example, if the minimal granularity of time is hour, then objects moving within the same hour can be seen as moving together and be merged into one movement. Similarly, if the granularity of the location is shelf, objects moving to the different layers of a shelf can be seen as moving to the same *shelf* and be merged into one.

Another type of path generalization, is that of expanding different types of locations to different levels of abstraction depending on the analysis task. For example, a transportation manager my want to collapse all movements inside

stores, and warehouses, while expanding movements within trucks and transportation centers to a very detailed level. On the other hand, store managers may want to collapse all object movements outside their particular stores.

An important difference between path level generalization and the more conventional data cube generalization along concept hierarchies, is that in path level aggregation we need to preserve the path structure of the data, *i.e.*, we need to make sure that the new times, locations, and transitions are consistent with the original data.

Item Level Generalization

This type of generalization is the one encountered in traditional data cubes, and it is done on the non spatio-temporal dimensions used to describe items, such as, product, manufacturer, or price. Each of these dimensions usually involves a concept hierarchy, and OLAP would rarely take place at the lowest abstraction level. This is especially important for RFID applications, where data at the EPC level can be very large. For example, in most analysis tasks, it is not necessary to distinguish two cans of soda with the same Stock Keeping Unit (SKU).

In our proposed model we will allow for analysis to be conducted on highly compressed data, but will still allow for drilling to particular EPC tags when necessary.

4.3 Movement Graph Partitioning

In this section we discuss the methods for identifying gateways, partitioning based on the *movement graph*, and associating partitions to gateways.

4.3.1 Gateway Identification

In many applications it is possible for data analysts to provide the system with the complete list of gateways, this is realistic in a typical supply-chain application where the set of transportation ports is well known in advance, *e.g.*, Walmart knows all the major ports connecting its suppliers in Asia to the entry ports in the United States. In some other cases, we need to discover gateways automatically. We can use existing graph partitioning techniques such as *balanced minimum cut* or *average minimum cut* [21], to find a small set of edges that can be removed from the graph so that the graph is split into two disconnected components; such edges will typically be associated with the strong traffic edges of in- or out- gateways. Gateways could also be identified by using the concept of betweenness and centrality in social network analysis as they will correspond to nodes with high betweenness as defined in [33] and we can use an efficient algorithm such as [10] to find them.

Here we propose a simple but effective approach to discover gateways, that works well for typical supply-chain operations where gateways have strong characteristics that are easy to identify. We can take a *movement graph*, and

rank nodes as potential gateways based on the following observations: (i) a large volume of traffic goes through gateway nodes, (ii) gateways carry *unbalanced traffic*, *i.e.*, incoming and outgoing edges carrying the same tags but having very different average shipment sizes, and (iii) gateways split paths into largely disjoint sets of nodes that only communicate through the gateway. The algorithm can find gateways by eliminating first low traffic nodes, and then the nodes with balanced traffic, *i.e.*, checking the number of incoming and outgoing edges, and the ratio of the average incoming (outgoing) shipment sizes vs. the average of the outgoing (incoming) shipment sizes. Finally, for the edges that pass the above two filters, check which locations split the paths going through the location into two largely disjoint sets. That is, the locations in paths involving the gateway can be split into two subsets, locations occurring in the path before the gateway and those occurring in the path after the gateway.

4.3.2 Partitioning Algorithm

The movement graph partitioning problem can be framed as a traditional graph clustering problem and we could use techniques such as spectral clustering [52, 21]. But for the specific problem of partitioning supply-chain *movement graphs* we can design a less costly algorithm that takes advantage of the topology of the graph to associate locations to those gateways to which they are more strongly connected.

The key idea behind the partitioning algorithm is that in the movement graph for a typical supply chain application locations only connect directly (without going through another gateway) to a few gateway nodes. That is, very few items in Europe reach the major ports in the United States without first having gone through Europe's main shipping ports. Using this idea, we can associate each location to the set of gateways that it directly reaches (we use a frequency threshold to filter out gateways that are reached only rarely), when two locations l_i and l_j have a gateway in common we merge their groups into a single partition containing the two locations and all their associated gateways. We repeat this process until no additional merge is possible. At the end we do a postprocessing step where we associate very small partitions to the larger partition to which it most frequently directly connects to.

Analysis. Algorithm 1 presents the details of *movement graph* graph partitioning given a set of gateways. In a single scan of the path database, we compute statistics on the traffic from each node to the different gateways. We then go through the list of locations merging sets of locations that share common gateways. Finally, we merge small clusters into larger ones. This algorithm scales linearly with the size of the path database, linearly with the number of nodes in the *movement graph*, and quadratically with the number of gateways in the *movement graph*. We can further speed up the algorithm by running it on a random sample of the original database instead of running it on the full data. This is possible because the structure of the supply chain is usually fairly stable over time, and a representative random sample is enough to capture the topology of the graph.

Algorithm 1 Movement graph partitioning

Input: $G(V, E)$: a movement graph, $W \subset V$: the set of gateways, D: a path database, min_nodes: min. # of vertices per partition, $min\_connectivity$: min. # of paths to gateway.
Output: A partition of V into $V_1, \ldots, V_k$ s.t. $V_i \bigcap V_j = \phi, \forall i \neq j$.
Method:

1: Let $\mathcal{C}$ be a connection matrix, with entries $\mathcal{C}[l_i, g_j]$ that indicate the number of times that location l_i connects to gateway g_j. Initialize every entry in $\mathcal{C}$ to 0.
2: **for** each path p in D' **do**
3: **for** each location l in p **do**
4: $\mathcal{C}[l, g] += 1$ where g is the next gateway after l in p, or g is the previous gateway before l in p.
5: **end for**
6: **end for**
7: **for** each location $l \in V$ **do**
8: $L_g =$ all gateways s.t. $\mathcal{C}[l, g] > min\_connectivity$
9: add l to the partition containing all gateways in L_g, if the gateways in L_g reside in different partitions merge the partitions and add l and the the merged partition, if no partition exists that contains any element in L_g create a new partition containing $L_g \bigcup \{l\}$.
10: **end for**
11: **for** all partitions V_i s.t. $|V_i| < min\_nodes$ **do**
12: merge V_i with the partition V_j s.t. the traffic from/to gateways in V_i to/from gateways in V_j is maximum, this can be determined using the connection matrix $\mathcal{C}$.
13: **end for**
14: return partitions $V_1, ..., V_k$

4.3.3 Handling Sporadic Movements: Virtual Gateways

An important property of gateways is that all the traffic leaving or entering a partition goes through them. However, in reality it is still possible to have small sets of sporadic item movements between partitions that bypass gateways. Such movements reduce the effectiveness of gateway-based materialization because path queries involving multiple partitions will need to examine some path segments of the graph unrelated to gateways. This problem can be easily solved by adding a special *virtual gateway* to each partition for all outgoing and incoming traffic from and to other partitions that does not go through a gateway. Virtual gateways guarantee that inter-gateway path queries can be resolved by looking at gateway-related traffic only.

For our running example, Figure 4.2, we can partition the *movement graph* along the dotted circles, and associate gateway G_1 with the first partition, and gateway G_2 with the second one. In this case we need to create a virtual gateway G_x to send outgoing traffic from the first partition (*i.e.* traffic from B) that skips G_1, and another virtual gateway G_y to receive incoming traffic into the second partition (*i.e.* traffic to I), that skips G_2.

4.4 Storage Model

With the tremendous amounts of RFID data, it is crucial to study the storage model. We propose to use three data structures to store both compressed and generalized data: (1) an edge table, storing the list of edges, or alternatively

an stay table, storing the list of nodes (2) a map table, linking groups of items moving together, and (3) an information table, registering path-independent information related to the items in the graph.

4.4.1 Edge Table

This table registers information on the edges of the *movement graph*, the format is $\langle from, to, t\_start, t\_end, direct, gid\_list, : measure\_list\rangle$, where $from$ is the originating node, to is the destination node, t_start is the time when the items departed the location $from$, t_end is the time when the items arrived at the location to, $direct$ is a boolean value that is true if the items moved directly between $from$ and to and false if intermediate nodes were visited, gid_list is the list of items that traveled together, and $measure\_list$ is a set of aggregate functions computed on the items in gid_list while they took the transition, *e.g.*, it can be count, average temperature, average movement cost, *etc.*. We will elaborate more on concept of $gids$ in the section describing the map table.

4.4.2 Stay Table

An alternative to recording edges in the graph is to record nodes, and the history of items staying at each node. The table register $\langle location, time\_in, time\_out, gid\_list : measure\_list\rangle$, where $location$ is the location where the items were detected, $time\_in$ is the time when the items entered $location$, $time\_out$ is the time when the items left $location$, and the rest of columns are defined as before. In some applications, registering stay records may be useful, especially when most queries inquire about the properties of items as they stay at a particular location more than the properties of items as they move between locations.

In order to provide a uniform model, we can accommodate both views by just switching locations and edges in the *movement graph*, *i.e.*, in this view edges represent locations, and nodes represent transitions. We can also adapt the model for the case when we record item properties for both locations and transitions, by modifying the *movement graph* to associate both transitions and locations with edges. Which model to choose will depend on applications. For example, if the truck temperature during the transportation is essential, take the edge table. If storage room situation is critical, take the stay table. If both are important, use both tables. In the rest of this chapter we will assume, without loss of generality, that edges in the *movement graph* are associated with transitions, but it should be clear that all the techniques and algorithms would work under alternative edge interpretations.

4.4.3 Map Table

Bulky movement means a large number of items move together. A generalized identifier gid can be assigned to every group of items that moves together, which will substantially reduce the size of the tag_lists at each edge. When groups of items split into smaller groups, gid (original group) can be split into a set of children gids, representing

these smaller groups. The map table contains entries of the form $\langle partition, gid, contained\_list, contains\_list \rangle$, where $partition$ is the subgraph of the *movement graph* where this map is applicable, $contained\_list$ is the list of all $gids$ with a list of items that is a superset of the items gid, $contains\_list$ is the list $gids$ with item lists that are a subset of gid, or a list of individual tags if gid did not split into smaller groups.

There are two main reasons for using a *map* table instead of recording the complete EPC lists at each stage: (1) data compression, and (2) query processing efficiency.

Compression: First, we do not want to record each RFID tag on the EPC list for every *stay* record it participated in. For example, if we assume that 10000 items move in the system in groups of 10000, 1000, 100, and 10 through 4 stages, instead of using 40,000 units of storage for the EPCs in the *stay* records, we use only 1,111 units [2] (1000 for the last stage, 100, 10, and 1 for the ones before).

Since real RFID datasets involve both merge and split movement models, the *Split-Only* model cannot have much sharing and compression. Here we adopt a *Merge-Split* model, where objects can be merged, shuffled, and split in many different combinations during transportation. Our mapping table takes a *gateway-centered model*, where mapping is centered around gateways, *i.e.*, the largest merged and collective moving sets at the gateways become the root gids, and their children gids can be spread in both directions along the gateways. This will lead to the maximal gid sharing and $gid_$list compression.

Query Processing: The second and the more important reason for having such a map table is the efficiency in query processing. By creating gid lists that are much shorter than EPC lists, we can compute path related queries very quickly. To compute, for example, the average duration for milk to move from the distribution center (D), to the store backroom (B), and finally to the shelf (S), we need to locate the edge records for milk between the stages and intersect the EPC lists of each. By using the map, the EPC lists can be orders of magnitude shorter and thus reduce IO costs.

Path-Dependent gid naming.

In order to facilitate query processing we will assign path-dependent labels to high level $gids$. The label will contain one identifier per location traveled by the items in the gid. Figure 4.4 presents an example gid map. We see that the group of items with gid 0.0 arrive at l1, and then subdivide into 0.0.0, and s0.0, the items in s0.0 (we prefix the label with *s* to denote that the items stay in the location) stay in location l1 while the items in 0.0.0 travel to l3, and subdivide further into 0.0.0.0 and s0.0.0, the items in s0.0.0 stay at location l3, while the ones in 0.0.0.0 move to location l5.

[2] This figure does not include the size of the map itself which should use 12,221 units of storage, still much smaller than the full EPC lists

Map - Tabular View

0	0.0, 0.1
0.0	0.0.0,s0.0
0.0.0	0.0.0.0,s0.0.0
0.0.0.0	r2,r3
s0.0	r4
s0.0.0	r1
0.1	0.1.0,0.1.1
0.1.0	0.1.0.0,0.1.0.1
0.1.0.0	r5
0.1.0.1	r6
0.1.1	r7

Map - Graphical View

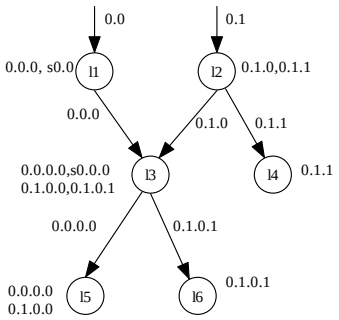


Figure 4.4: GID Map - tabular and graphical views

4.4.4 Information Table

The information table records other attributes that describe properties of the items traveling through the edges of the *movement graph*. The format of the tuples in the information table is $\langle gid\_list, D_1, \ldots, D_n \rangle$, where gid_list is the list of items that share the same values on the dimensions D_1 to D_n, and each dimension D_i describes a property of the items in gid_list. An example of attributes that may appear in the information table could be $product$, $manufacturer$, or $weight$. Each dimension of the information table can have an associated concept hierarchy, *e.g.*, the $product$ dimension may have a hierarchy such as $EPC \rightarrow SKU \rightarrow product \rightarrow category$.

4.5 Materialization Strategy

Materialization of path segments in the *movement graph* may speedup a large number of path-related queries. Since there is an exponential number of possible path segments that can be pre-computed in a large *movement graph*, it is only realistic to partially materialize only those path segments that provide the highest expected benefit at a reasonable cost. We will develop such a strategy here.

4.5.1 Path Queries

A path query requires the computation of a measure over all the items with a path that matches a given *path pattern*. It is of the form: $q \leftarrow \langle \sigma_c info, path\_expression, measure \rangle$, where $\sigma_c info$ is a selection on the information table that retrieves the relevant items for analysis; $path\_expression$ is a sequence of stage conditions on location and time that should appear in every path, in the given order but possibly with gaps; and $measure$ is a function to be computed on the matching paths. An example path query may be $\sigma_c info$ = {product = meat, sale_date = 2006}, $path\_expression$ = {Argentina farm A, San Mateo store S}, and $measure = average\ temperature$, which asks for the average temperature recorded for each meat package, traveling from a certain farm in Argentina to a particular

store in San Mateo.

There may be many strategies to answer a path query, but in general we will need to retrieve the appropriate tag lists and measures for the edges along the paths involving the locations in the *path expression*; retrieve the tag list for the items matching the info selection; intersect the lists to get the set of relevant tags; and finally, if needed, retrieve the relevant paths to compute the measure.

Using the path dependant naming of gids we can further optimize query processing. Given a path query, or a segment of the path query that involves a single partition of the movement graph, and path dependent gid naming, we can use the following process. (1) Select the $gids$ for the edge records that match the conditions for the initial and final stages of the query expression and store them in g_{start}, and g_{end} respectively. For example, g_{start} may look like $\langle 1.2, 8.3.1, 3.4 \rangle$ and g_{end} may look like $\langle 1.2.4.3, 4.3, 3.4.3 \rangle$. (2) Compute the pairs of gids from g_{start} that are a prefix of a gid in g_{end}. Continuing with the example we get the pairs $\langle (1.2, 1.2.4.3), (3.4, 3.4.3) \rangle$. (3) For each pair we then retrieve all the relevant edge records. The pair $(1.2, 1.2.4.3)$ would require us to retrieve records that include gids 1.2, 1.2.4, and 1.2.4.3. (4) Verify that each of these records matches the selection conditions for each $stage_i$ and for $info$, and add those paths to the answer set.

4.5.2 Path-segment Materialization

We can model path segments as indirect edges in the *movement graph*. For example, if we want to pre-compute the list of items moving from location l_i to location l_j through any possible path, we can materialize an edge from l_i to l_j that records a history of all tag movements between the nodes, including movements that involve an arbitrary number of intermediate locations. Indirect edges are stored in the same format as direct ones, but with the flag $direct$ set to $false$.

The benefit of materializing a given indirect edge in the *movement graph* is proportional to the number of queries for which this edge reduces the total processing cost. Indirect edges, involved in a path query, reduce the number of edges that need to be analyzed, and provide shorter tag lists that are faster to retrieve and intersect. In order for an indirect edge to help a large number of queries, it should have three properties: (1) carry a large volume of traffic, (2) be part of a large portion of all the paths going from nodes in one partition of the graph to nodes in any other partition, and (3) be involved directly or indirectly in a large number of path queries. The set of edges that best match these characteristics are the following.

Node-to-gateway

. In supply-chain implementations it is common to find a few well defined *Out-gateways* that carry most of the traffic leaving a partition of the graph where items are produced, before reaching a partition of the graph where items are

consumed. For example, products manufactured in China destined for exports to the United States leave the country through a set of ports. We propose to *materialize the (virtual) edges from every node to the Out-gateways that it first reaches*. Such materialization, for example, would allow us to quickly determine the properties of shipments originating at any location inside China and leaving the country.

Gateway-to-node

. Another set of important nodes for indirect edge materialization are *In-gateways*, as most of the item traffic entering a region of the graph where items are consumed has to go through an *In-gateway*. For example, imported products coming into the United States all arrive through a set of major ports. When we need to determine which items sold in the U.S. have paths that involve locations in foreign countries, we can easily get this information by pre-computing the list of items that arrived at the location from each of the *In-gateways*. We propose to *materialize all the (virtual) edges from an* In-gateway *to the nodes that it reaches without passing through any other gateway*.

Gateway-to-gateway

. Another interesting set of indirect edges to materialize are *the ones carrying inter-gateway traffic*. For example, we want to pre-compute which items leaving the Shanghai port finally arrive at the New York port. The benefit of such indirect edge is twofold: First, it aggregates a large number possible paths between two gateways and pre-computes important measures on the shipments; and second, it allows us to quickly determine which items travel between partitions.

Lemma 4.5.1 *A* movement graph *with k partitions, $p_1, ..., p_k$, each partition i with p_i^n nodes and p_i^g gateways, will require the materialization of a number of indirect edges that is bounded by $\sum_{i=1}^{k} (p_i^n \times p_i^g) + \sum_{i \neq j} p_i^g \times p_j^g$*

Proof. For each node in each partition we will materialize at most p_i^g indirect edges, this is when the node has traffic to or from every gateway, the maximum number of node to gateway edges is then $\sum_{i=1}^{k} p_i^n \times p_i^g$, the maximum number of gateway to gateway edges occurs when every gateway has traffic to every other gateway for a maximum number of gateway to gateway edges that is $\sum_{i \neq j} p_i^g \times p_j^g$ ∎

The implication of lemma 4.5.1 is that the size of our materialization scheme is small in size, especially when compared to a full materialization model in which we compute direct edges between every pair of nodes in which case we would require $(\sum_{i=1}^{k} p_i^n)^2$ additional edges. In practice the overhead of gateway materialization is usually smaller than the bound found in lemma 4.5.1, the reason is that nodes in a partition will tend to associate with a single gateway, and partitions will connect to a small subset of other partitions.

Lemma 4.5.2 *Given a* movement graph *with k partitions, $p_1, \ldots, p_k$, each partition i with p_i^n nodes and p_i^g gateways, any pairwise path query q involving a path expression with two nodes (n_i, n_j), where $n_i \in p_i$, $n_j \in p_j$, can be answered by analyzing at most $p_i^g + p_j^g + p_i^g \times p_j^g$ edges.*

Proof. In the worst case traffic can travel from node n_i to node n_j through all the gateways in partition p_i and all the ones in partition p_j so we need to analyze at most $p_i^g + p_j^g$ node to/from gateway indirect edges, in the worst case traffic can travel between any pair of gateways in p_i and p_j for a maximum number of inter-gateway indirect edges of $p_i^g \times p_j^g$. ∎

Lemma 4.5.2 provides a bound on the cost of query processing when gateway materialization has been implemented.

In general, when we need to answer a path query involving all paths between two nodes, we need to retrieve all edges between the nodes and aggregate their measures. This can be very expensive if the locations are connected by a large number of distinct paths, which is usually the case when nodes are in different partitions of the graph. By using gateway materialization we reduce this cost significantly as remote nodes can always be connected through a few edges to, from, and between gateways.

4.6 RFID Cube

So far we have examined the *movement graph* at a single level of abstraction. Since items, locations (as nodes in the graph), and the history of shipments along each edge all have associated concept hierarchies, aggregations can be performed at different levels of abstraction. We propose a data cube model for such *movement graphs*. The main difference between the traditional cube model and the *movement graph* cube is that the former aggregates on simple dimensions and levels but the latter needs to aggregate on path-dimensions as well, which may involve path collapsing as a new form of generalization. In this section we develop a model for *movement graph* cubes and introduce an efficient algorithm to compute them.

4.6.1 Movement Graph Aggregation

With a concept hierarchy associated with locations, a path can be aggregated to an abstract location by aggregating each location to a generalized location, collapsing the corresponding movements, and rebuilding the *movement graph* according to the new path database.

Location aggregation. We can use the location concept hierarchy to aggregate particular locations inside a store to a single store location, or particular stores in a region to a single region location. We can also completely disregard

certain locations not interesting for analysis, *e.g.*, a store manager may want to eliminate all factory-related locations from the *movement graph* in order to see a more concise representation of the data.

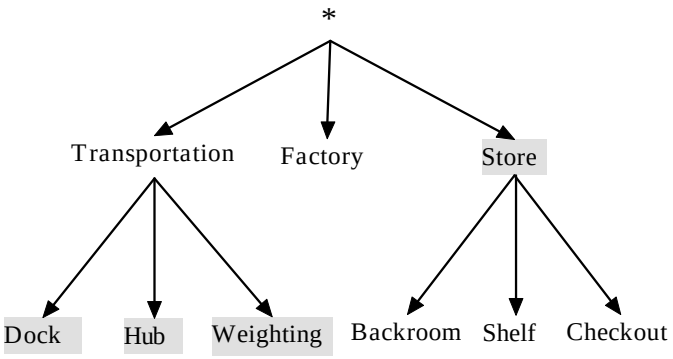


Figure 4.5: Location concept hierarchy

Figure 4.6 presents a *movement graph* and some aggregations. All the locations are initially at the lowest abstraction level. By generalization, the transportation-related locations are collapsed into a single node $Transportation$, and the store-related locations into a single node $Store$ (shown as dotted circles). Then the original single path: $Factory \rightarrow Dock \rightarrow Hub \rightarrow Backroom \rightarrow shelf$ is collapsed to the path $Factory \rightarrow Transportation \rightarrow Store$ in the aggregated graph. If we completely remove transportation locations, we will get the path $Factory \rightarrow Store$.

Edge aggregation semantics. From the point of view of the edge table, graph aggregation corresponds to merging of edge entries, but it is different from regular grouping of fact table entries in a data cube because collapsing paths will create edges that did not exist before, and some edges can be completely removed if they are not important for analysis. In a traditional data cube, fact table entries are never created or removed, they are just aggregated into larger or smaller groups. For example, in Figure 4.6, if we remove all transportation-related locations, a new edge ($Factory$, $Store$) will be created, with all the edges to and from transportation locations removed.

Graph aggregation involves different operations over the gid_lists at each edge, when we remove nodes we need to intersect gid_lists to determine the items traveling through the new edge, but when we simply aggregate locations to higher levels of abstraction (without removing them) we need instead to compute the union of the gid_lists of several edges. For example, looking at Figure 4.6 in order to determine the gid_list for the edge ($Factory$, $Store$) we need to intersect the gid_lists of all outgoing edges from the node $Factory$ with the incoming edges to the node $Store$; on the other hand if we aggregate transportation locations to a single node, in order to determine the gid_list for the edge ($Transportation$, $Store$), we need to union the gid_lists of the edges $(Hub, Store)$ and $(Weighting, Store)$.

Item aggregation. Each item traveling in the *movement graph* has an associated set of dimensions that describe its properties. These dimensions can also be aggregated, *e.g.*, an analyst may ask for the *movement graph* of *laptops* in

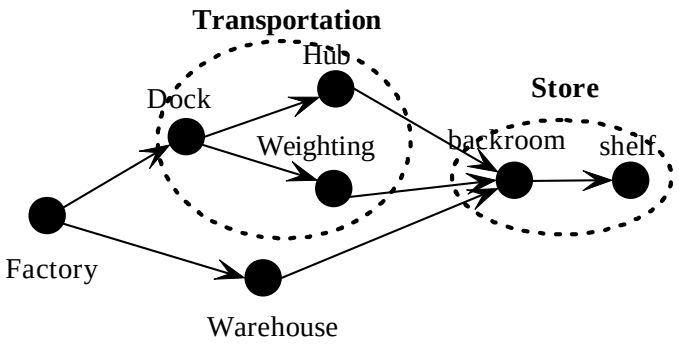


Figure 4.6: Graph aggregation

March 2006, while another analyst may be interested in looking at more general movements for *electronic goods* in *2006*. Aggregation along info dimensions is the same type of aggregation that we see in traditional data cubes, we group items according to the values they share at a given abstraction level of the info dimensions.

There are thus, two distinct but related views on *movement graph* aggregation. The first view, which we call *path aggregation* corresponds to the merging of nodes in the *movement graph* according to the location concept hierarchy. The second view, corresponds to aggregation of shipments and items traveling through the edges of the *movement graph*, according concept hierarchies for time and info dimensions.

4.6.2 Cube Structure

Fact table. The fact table contains information on the *movement graph*, and the items aggregated to the minimum level of abstraction that is interesting for analysis. Each entry in the fact table is a tuple of the form: $\langle from$, to, t_start, t_end, d_1, d_2, ..., $d_k : gid\_list : measure\_list\rangle$, where gid_list is list of $gids$ that contains all the items that took the transition between $from$ and to locations, starting at time t_in and ending at time t_out, and all share the dimension values d_1, ..., d_k for dimensions D_1, ..., D_k in the info table, $measure\_list$ contains a set of measures computed on the $gids$ in gid_list.

Measure. For each entry in the fact table we register the gid_list corresponding to the tags that match the dimension values in the entry. We can also record for each gid in the list a set of measures recorded during shipping, such as average temperature, total weight, or count. We can use the gid_list to quickly retrieve those paths that match a given *slice*, *dice*, or *path selection* query at any level of abstraction. When a query is issued for aggregate measure that are already pre-computed in the cube, we do not need to access the path database, and all query processing can be done directly on the aggregated *movement graph*. For example, if we record $count$ as a measure, any query asking for counts of items moving between locations can be answered directly by retrieving the appropriate cells in the cube. When a query asks for a measure that has not been pre-computed in the cube, we can still use the aggregated cells to

quickly determine the list of relevant $gids$ and retrieve the corresponding paths to compute the measure on the fly.

RFID-cuboids. A cuboid in the RFID cube resides at a level of abstraction of the location concept hierarchy, a level of abstraction of the time dimension, and a level of abstraction of the *info dimensions*. *Path aggregation* is used to collapse uninteresting locations, and *item aggregation* is used to group-related items. *Cells* in a *movement graph* RFID-cuboids group both items, and edges that share the same values at the RFID-cuboid abstraction level.

It is possible for two separate RFID-cuboids to share a large number of common cells, namely, all those corresponding to portions of the *movement graph* that are common to both RFID-cuboids, and that share the same *item aggregation* level. A natural optimization is to compute cells only once. The size of the full *movement graph* cube is thus the total number of distinct cells in all the RFID-cuboids. When materializing the cube or a subset of RFID-cuboids we compute all relevant cells to those RFID-cuboids without duplicating shared cells between RFID-cuboids.

4.6.3 Cube Computation

In this section we introduce an efficient algorithm, that in a single scan of the fact table, simultaneously computes the set of interesting RFID-cuboids, as defined by the user or determined through selective cube materialization techniques such as those proposed in [52]. The computed RFID-cuboids can then be used to answer the most common OLAP and path query operations efficiently, and can also be used to quickly compute non-materialized RFID-cuboids on the fly.

Partition-based aggregation. We will aggregate each partition of the graph independently, *i.e.*, paths will be divided into disjoint segments according to the partitions defined in the *movement graph*, and each segment in the path will be aggregated independently, without merging locations from separate segments. This technique guarantees that for any RFID-cuboid we can still use the gateway-based materialization to improve query performance. If we need to compute inter-partition aggregation, it can be done at runtime by using the best available RFID-cuboids from each partition, and computing required aggregation on top of those at runtime.

Algorithm 2 presents an efficient cubing algorithm that does simultaneous aggregation of every interesting cell in parallel, with a single scan of the path database.

Path prefix tree. The algorithm first constructs a prefix tree for each partition of the *movement graph*. For this purpose paths are divided into disjoint fragments separated by gateways, in the case when locations belonging to separate partitions appear in the same path fragment we separate them with a virtual gateway. Each path is converted into a sequence of edges of the form $(from, to, t\_in, t\_out)$, the first edge of every path has a $from$ location equal to the special symbol $\vdash$, and the last edge in the path has a to location that is the special symbol $\dashv$. After the prefix trees have been constructed, we assign a unique gid to the items aggregated in each node of the tree, that share the same

values on all *item dimensions*.

Figure 4.7 presents the path prefix tree computed on the path database of Table 4.1, and the first partition in Figure 4.2. Notice that we have created the virtual gateway G_x and added it to the prefix path. Figure 4.8 presents the prefix path tree for the second partition, again we have created an edge from virtual gateways G_x to G_y to capture those items not traveling through ordinary gateways.

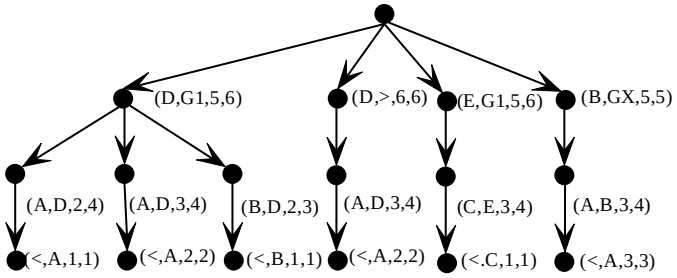


Figure 4.7: Prefix tree partition 1

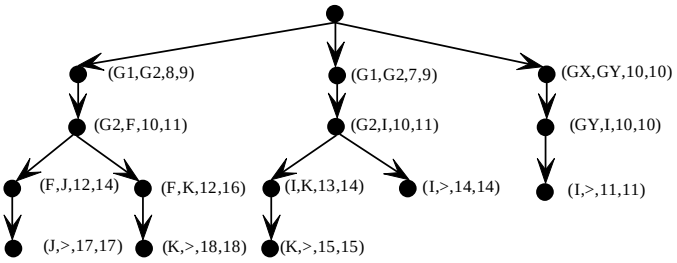


Figure 4.8: Prefix tree partition 2

Cuboid materialization. After building the path prefix trees, we are ready to start building the relevant set of RFID-cuboids, this is done by traversing each prefix tree, one branch at a time, and generating all possible edges from the branch, including: (1) direct edges that correspond to a single node in the tree, (2) edges generated by performing *path collapsing* to all interesting location levels of abstraction, and (3) indirect edges from each location in the path to a gateway. All the cells involving these edges are then updated to include the $gids$ associated with the edge and their measures adjusted. We also do aggregation of RFID-cuboids that include only *item dimensions* (*i.e.*, location and time dimensions are aggregated to *all*) by aggregating the *info* entry for the relevant $gids$ to the levels indicated by the set of RFID-cuboids to compute. The simultaneous multi-level, multi-dimensional aggregation is similar to the aggregation done in multi-way array aggregation [94], and we can make use of very similar techniques to minimize the amount of memory required to materialize the cube; the idea is to sort the dimensions of the info table in decreasing cardinality and to build the prefix path tree in lexicographic order, so that we minimize the number of cells that need to be kept in

main memory.

Table 4.2 presents an example RFID-cuboid computed on the first partition of the *movement graph* of Figure 4.2. In this RFID-cuboid locations C and D are uninteresting, and both time and item dimensions have been aggregated to *all*. Each entry is a cell, and the measure is a tag_list[3].

from	to	t1	t2	product	tag_list
$\vdash$	A	*	*	*	t1,t2,t3,t6
$\vdash$	B	*	*	*	t4
$\vdash$	C	*	*	*	t5
A	G_1	*	*	*	t1,t2
A	G_x	*	*	*	t6
A	$\dashv$	*	*	*	t3
B	G_1	*	*	*	t4
B	G_x	*	*	*	t6
C	G_1	*	*	*	t5

Table 4.2: RFID-cuboid example

Analysis. Algorithm 2 can be implemented efficiently, as it requires a single scan of the path database, which is compressed into a compact prefix tree representation, which in turn is traversed only once to generate all relevant aggregated cells in parallel. It is more efficient than the cubing algorithm proposed in [42] which computes RFID-cuboids one at a time and thus misses the efficiency gains provided by shared computation of multiple path segments in parallel. [42] also incurs in the cost of maintaining a separate info and map table for every RFID-cuboid, we instead keep a single map table for all RFID-cuboids, and incorporate the item dimensions in the graph itself, so no extra overhead of accessing the info table is required when slicing on item dimensions.

4.7 Experimental Evaluation

In this section we report our comprehensive evaluation of the proposed model and algorithms. All the experiments were conducted on a Pentium 4 3.0 GHz, with 1.5Gb RAM, running Win XP; the code was written in C++ and compiled with Microsoft Visual Studio 2003.

4.7.1 Data Synthesis

The path databases used for performance evaluation were generated using a synthetic path generator. We first generate a random *movement graph* with 5 partitions and 100 locations, each partition has a random number of gateways. Locations inside a partition are arranged according to a *producer configuration*, where we simulate factories connecting to intermediate locations that aggregate traffic, which in turn connect to *Out-gateways*; or a *consumer configuration*,

[3]In reality this list would be a gid_list, but we use tag_list in order to make the example easier to understand

Algorithm 2 Graph cube construction

Input: D: path database, P: location partitions, W: set of gateways for each partition, and $C = \{c_1, \ldots, c_m\}$: set of interesting RFID-cuboids to materialize
Output: The cells for the RFID-cuboids in C, and a map table
Method:

1: Scan D once and build a prefix tree of edges for each partition. Each path in D is broken into partitions according to P and W, if needed virtual gateways are inserted to separate partitions. Paths are aggregated to the minimum interesting abstraction level. Nodes in the prefix trees have the form $(from, to, t\_in, t\_out)$. Prefix trees should always be rooted at the gateways.
2: **for** each prefix tree T_i **do**
3: Assign $gids$ to each node in the tree, by linking each gid to all of its direct children in the tree, and the complete list of ancestors
4: **for** each branch p in T_i **do**
5: Let p_d be the set of direct edges
6: compute p_i the set of indirect edges created by path collapsing according to the location abstraction level of RFID-cuboids in C
7: compute p_g the set of node to/from gateway edges
8: aggregate each relevant cell using all elements in p_d, p_i, and p_g.
9: aggregate cells involving only *item dimensions*
10: **end for**
11: **end for**

where we simulate products moving from *In-gateways*, to intermediate locations such as distribution centers, and finally to stores. We generate paths by simulating groups of items moving inside a partition, or between partitions, and going usually through gateways, but sometimes, also "jumping" directly between non-gateway nodes; we increase shipment sizes close to gateways. We control the number of items moving together by a *shipment size* parameter, which indicates the smallest number of items moving together in a partition. Each item in the system has an entry in the path database, and an associated set of *item dimensions*. We characterize a dataset by $\mathcal{N}$ the number of paths, and $\mathcal{S}$ the minimum shipment size.

In most of the experiments in this section we compare three competing models. The first model is *part gw mat*, it represents a partitioned *movement graph* where we have performed materialization of path segments to, from, and between gateways. The second model is *part gw no mat*, it represents a model where we partition the *movement graph* but we do not perform gateway materialization. The third model is *no part*, which represents a *movement graph* that has not been partitioned and corresponds to the model introduced in [42].

4.7.2 Model Size

In these experiments we compare the sizes of three models of *movement graph* materialization, and the size of the original path database *path db*. For all the experiments, we materialize the graph at the same level of abstraction as the one in the path database and thus is a lossless representation of the data.

Figure 4.9 presents the size of the four models on path databases with a varying number of paths. For this ex-

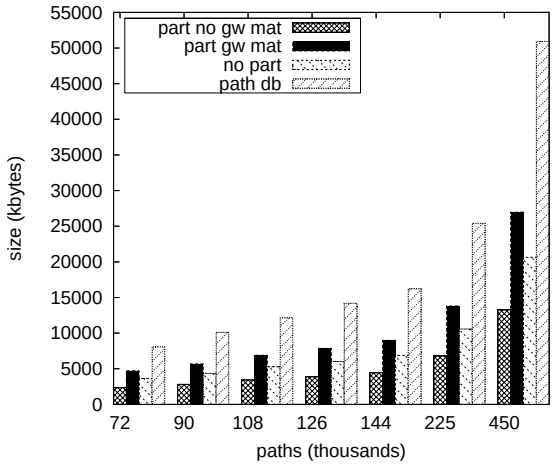


Figure 4.9: Fact table size vs. Path db size ($\mathcal{S} = 300$)

periment we can clearly see that the partitioned graph without gateway materialization *part no gw mat* is always significantly smaller than the non partition model *non part*, this comparison is fair as both models materialize only direct edges. When we perform gateway materialization the size of the model increases, but the increase is still linear on the size of the *movement graph* (much better than full materialization of edges between every pair of nodes which is quadratic in the size of the *movement graph*), and close to the size of the model in [42].

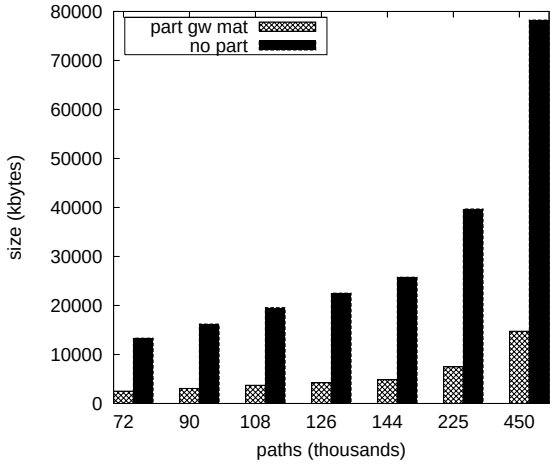


Figure 4.10: Map table size vs. Path db size ($\mathcal{S} = 300$)

Figure 4.10 presents the size of the map table for the partitioned *part gw mat* and non-partitioned *no part* models. The difference in size is almost a full order of magnitude. The reason is that our partition level maps capture the semantics of collective object movements much better than [42]. This has very important implications in compression power, and more importantly, in query processing efficiency.

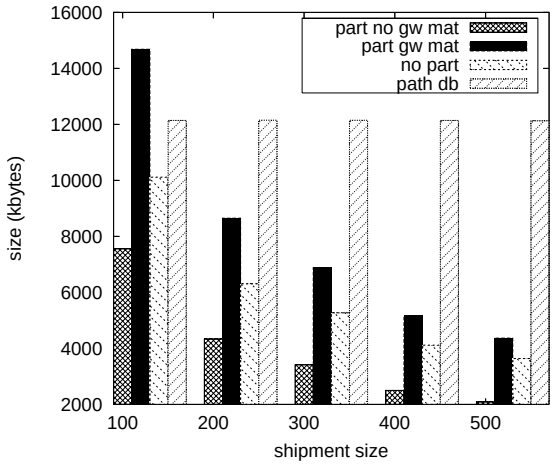


Figure 4.11: Fact table size vs. Shipment size ($\mathcal{N} = 108,000$)

Figure 4.11 presents the size of the fact table, as we vary the shipment size, under four different models *part no gw mat*, *part gw mat*, *no part*, and *path db*. We see that compression improves as we increase shipment sizes. Gateway materialization increases the size of the fact table, it is still much smaller than the original path database, except for very small shipment sizes, and it is also smaller than a non-partitioned fact table.

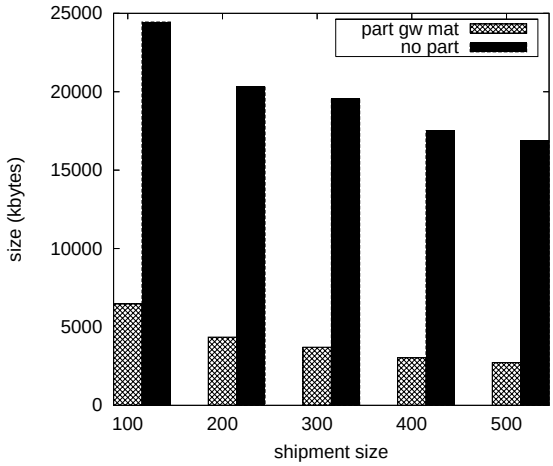


Figure 4.12: Map table size vs. Shipment size ($\mathcal{N} = 108,000$)

Figure 4.12 present the size of the map table, as we vary shipment size, for the *part gw mat* and *no part* models. This experiment isolates the effect on compression of the map table, due to shipment size. We can observe a very significant reduction in map size, even for small shipment sizes. This is a clear indication that partition level map tables provide a clear advantage over a global map table. This is important not only in terms of space, but also in terms of query processing, as we will see in the next section.

4.7.3 Query Processing

An important contribution of our model is efficiency in query processing. In these experiments we generate 100 random path queries that, ask for a measure on the path segments, for items matching an *item dimension* condition, that go from a single initial location to a single ending location, and that occurred within a certain time interval. We compare the partition *movement graph* with gateway materialization *part gw mat*, against the partitioned graph *part no gw mat* without gateway materialization, and the non-partitioned graph *no part*. All the queries were answering a *movement graph* at the same abstraction level as the original path database. For lack of space we restrict the analysis queries with starting and ending locations in different partitions as those are in general more challenging to answer. Based on our experiments on single partition queries, our method has a big advantage given by their compact map tables.

For the case of non-partitioned graph, we use the same query processing algorithm presented in [42]. For the case of partitioned graph without gateway materialization, we retrieve all the relevant edges from the initial node to the gateways in its partition, edges between gateways, and edges from the gateways in the ending location's partition and the location. In this case we do not perform inter-gateway join of the gid lists, but the overhead of such join can be small if we keep an inter-gateway join table, or if our measure does not require matching of the relevant edges in both partitions. For the gateway materialization case, we retrieve only relevant gateway-related edges. For this method we compute the cost using tag lists instead of gid lists on materialized edges to and from gateways.

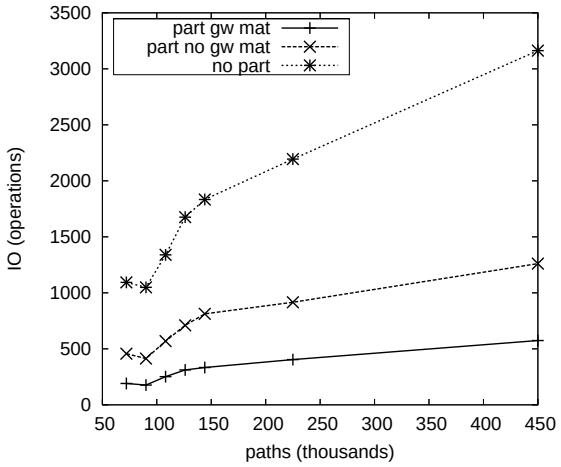


Figure 4.13: Query IO vs. Path db size ($\mathcal{S} = 300$)

In Figure 4.13 we analyze query performance with respect to path database size. We see that the gateway-based materialization method is the clear winner, its cost is almost an order of magnitude smaller than the method proposed in [42]. We also see that our method has the lowest growth in cost with respect to database size.

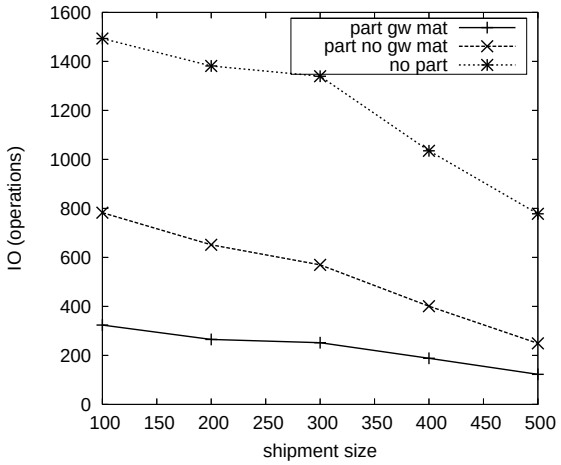


Figure 4.14: Query IO vs. Shipment size ($\mathcal{N} = 108,000$)

Figure 4.14 presents the same analysis but for a path database with different minimum shipment sizes. Our model is the clear winner in all cases, and as expected performance improves with larger shipment sizes.

4.7.4 Cubing

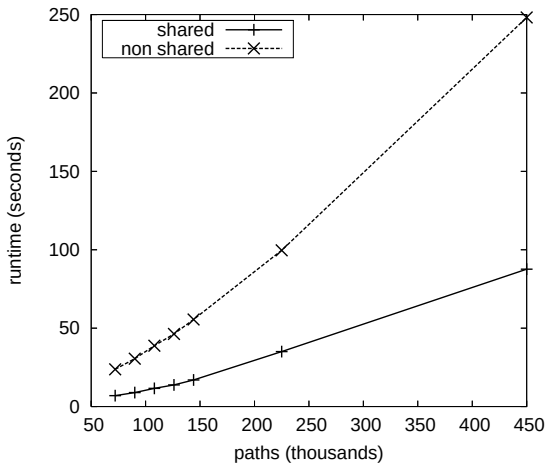


Figure 4.15: Cubing time vs. Path db size ($\mathcal{S} = 300\,\mathcal{N} = 108,000$)

For the cubing experiments we compute a set of 5 random cuboids, with significant shared dimensions among them, *i.e.*, the cuboids share a large number of interesting locations, and *item dimensions*. We are interested in the study of such cuboids because it captures the gains in efficiency that we would obtain if we used our algorithm to compute a full *movement graph* cube, as ancestor/descendant cuboids in the cube lattice benefit most from shared computation. Figure 4.15 presents the total runtime to compute 5 cuboids, we can see that *shared* significantly outperforms the level

by level cubing algorithm presented in [42]. For the case when cuboids are very far apart in the lattice the shared computation has a smaller effect and our algorithm performs similarly to [42].

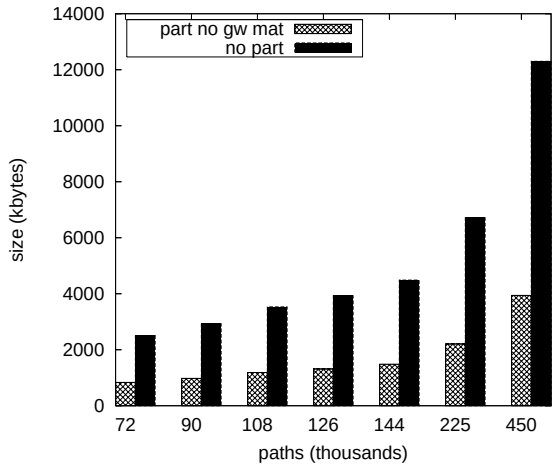


Figure 4.16: Cube size vs. Path db size ($\mathcal{S} = 300\,\mathcal{N} = 108,000$)

Figure 4.16 presents the total size of the cells in the 5 cuboids for the case of a partitioned graph without gateway materialization, and a non-partitioned graph. The compression advantage of our method increases for larger database sizes. This advantage becomes even more important as more cuboids are materialized. We can thus use our model to create compact *movement graphs* at different levels of abstraction, and furthermore, use them to answer queries significantly more efficiently than competing models. If we want even better query processing speed, we can sacrifice some compression and perform gateway-based materialization.

4.7.5 Partitioning

The final experiment evaluates the scalability of our *movement graph* partitioning algorithm. For this experiment we assume that the set of gateway nodes is given, which is realistic in many applications as this set is small and well-known (*e.g.*, major shipping ports). Figure 4.17 shows that the algorithm scales linearly with the size of the path database. And for our test cases, which are generated following the operation of a typical global supply chain operation, the algorithm always finds the correct partitions. This algorithm can partition very large datasets quickly, and at a fraction of the cost of standard graph clustering algorithms. It is important to note that for the case when we are dealing with a more general *movement graph*, that does not represent a supply chain, more expensive algorithms are likely required to find good partitions.

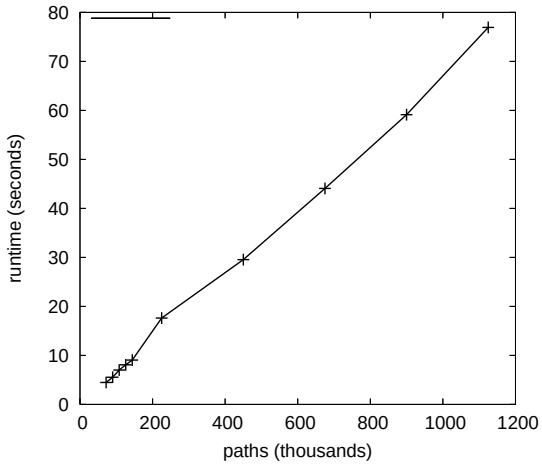


Figure 4.17: Partitioning time vs. Path db size

4.8 Summary

In this chapter we have introduced a new, *gateway-based movement graph model* for warehousing massive, transportation-based RFID datasets. This model captures the essential semantics of supply-chain application as well as many other RFID applications that explore object movements of similar nature. It provides a clean and concise representation of large RFID datasets. Moreover, it sets up a solid foundation for modeling RFID data and facilitates efficient and effective RFID data compression, data cleaning, multi-dimensional data aggregation, query processing, and data mining.

A set of efficient methods have been developed in this study for movement graph construction, gateway identification, gateway-based graph partitioning, efficient storage structuring, multi-dimensional aggregation, graph cube computation, and cube-based query processing. This weaves an organized picture for systematic modeling and implementation of such an RFID data warehouse. Our implementation and performance study shows that the methods proposed here are much more efficient in both storage cost, cube computation, and query processing comparing with a previous study [42] that uses a global map table without gateway-based movement graph modeling and partitioning.

The gateway-based movement graph model proposed here captures the semantics of bulky, sophisticated, but collective object movements, including merging, shuffling, and splitting processes. Its applications are not confined to RFID data sets but also extensible to other bulky object movement data. However, further study is needed to model and warehouse objects with scattered movements, such as traffic on highways where each vehicle moves differently from others. We are currently studying such modeling techniques and will report our progress in the future.

Chapter 5

RFID Data Cleaning

Efficient and accurate data cleaning is an essential task for the successful deployment of applications, such as object tracking and inventory management systems, based on RFID technology. Most existing data cleaning approaches do not consider the overall cost of cleaning in an environment that possibly includes thousands of readers and millions of tags. We propose a cleaning framework that takes an RFID data set and a collection of cleaning methods, with associated costs, and induces a *cleaning plan* that optimizes the overall *accuracy-adjusted cleaning costs*. The *cleaning plan* determines the conditions under which inexpensive cleaning methods can be safely applied, the conditions under which more expensive methods are absolutely necessary, and those cases when a combination of several methods is the optimal policy. Through a variety of experiments we show that our framework can achieve better accuracy at a fraction of the cost than that obtained by applying any single technique.

5.1 Error Sources in RFID Systems

There are two basic types of errors in reading tags: (1) *false negative*, which occurs when the reader does not detect a tag that is present within its detection range, and (2) *false positive*, which occurs when a reader detects a tag that should not be detected, for example, a reader located at a door detects a tag that goes close to the door but not through it. False positives can also occur as a byproduct of cleaning. For instance, if we use a smoothing window to detect a tag, a tag may be reported at a location mistakenly because of the smoothing.

In the rest of this section, we elaborate on the sources of both types of errors.

Collision Problems: These problems are usually caused by having multiple tags and/or readers transmitting at the same time. Collisions cause false negative errors. There are tag-tag collisions when multiple tags transmit simultaneously, reader-reader collision when the signals of two readers overlap, and reader-tag collisions when the signal of a reader collides with a response from a tag.

Environment interference: These problems are caused by objects in the reader's range that cause interference with the reader or tag signals. Common objects that cause interference are metal and water. RF noise caused by other equipments, such as fluorescent lamps and machinery, also generate interference.

Physical configuration problems: These problems are caused by the position or movement characteristics of tags with respect to the reader: (1) The tag is positioned such that it does not receive enough RF energy from the reader to power up and send its data. Some readers contain multiple antennas separated by one wavelength to reduce this problem; (2) The tag is moving too quickly, and there is no enough time for the reader to interrogate it and get a response; (3) The tag is positioned far away from the reader and the signal is too weak to be reliably detected every time.

Logical errors: These are not real errors in the sense that the hardware correctly detects a tag when it is within its range. These errors happen when business rules are violated, *e.g.*, when a reader located at a door is supposed to detect only tags that pass through the door, but it also detects tags that pass near the door, or when a driver goes through a door a couple of times because he forgot some documents inside the facility, which causes repeated readings of the tags. The correction of logical errors will depend on business rules defined by users of the system or learned from the data.

5.2 Cost-Conscious Cleaning

A cost-conscious approach to cleaning is to determine the context under which inexpensive methods work well, and try to clean as many RFID data as possible with such methods, while applying more expensive techniques only when absolutely necessary. For example, the system may automatically learn that for a particular reader the probability of detecting tags is almost constant and thus a fixed-window smoothing method is good enough, while it may detect that another reader, which covers a large area, requires variable-window smoothing. We may also learn that readings from generation 2 tags are reliable unless metal is present and thus we can trust the data coming from the reader in metal-free environments.

Figure 5.1 presents the architecture of the cleaning framework. We have a set of labeled data, which contain instances of tag readings annotated with features that describe the context in which the reading took place, and labeled with the cleaning methods that classify each case correctly. There is also a repository of available methods with cost information. The cleaning engine induces a cleaning plan that minimizes the cost of data cleaning while maintaining high accuracy, and uses this cleaning plan to clean the RFID stream. In the next few sections we will describe in detail each component of the framework.

Example Table 5.1 presents an example of labeled tag readings, with the correct location of the tag, and the cleaning results of applying three methods, *fix_1* which is a fixed-window method with window size 1, *dbn* which is a DBN-based method, and *pat* is a pattern matching method. From this example we see that *pat* works well on doors, as it correctly determines which read patterns correspond to items that actually go through the door and not near to the

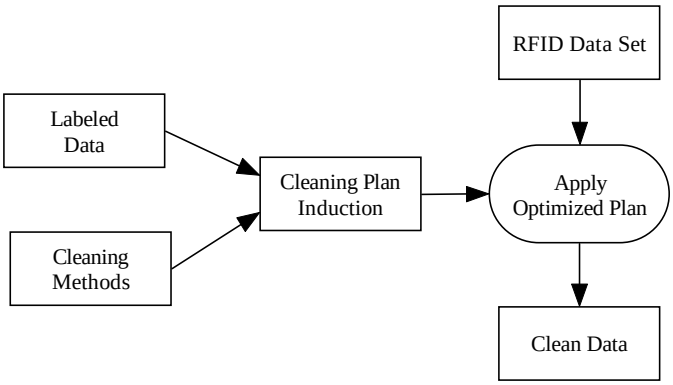


Figure 5.1: Architecture of the cleaning framework

cycle	tag	reader	metal	fix_1	dbn	pat	label
1	1	shelf	no	1	0	0	1
1	2	shelf	no	0	0	1	0
1	3	shelf	yes	0	1	0	1
2	4	shelf	no	0	0	0	0
2	5	shelf	yes	0	1	0	1
2	6	shelf	no	0	1	0	1
3	7	door	yes	1	1	0	0
3	8	door	no	1	1	0	0
3	9	door	yes	1	1	1	1

Table 5.1: Labeled tag cases

door. We also see that *fix_1* works well for items that contain no metal, and *dbn* works well on any reader that is not at a door. Also assume that each method has an associated cost to clean each case which is: *dbn*: 2.0, *fix*: 1.0, and *pat*: 2.0. And we also have a cost for each mistake that me make in cleaning a case which is 5.0.

5.2.1 Cleaning Methods

Cleaning of RFID data can be seen as a classification problem. We take as input a set of features describing the current information regarding a particular tag, and decide on its location. For example, a cleaning method may use the features $(EPC, reader, timestamp, detected)$, where $detected$ is a binary flag, that is set to 1 if the $reader$ detected the tag identified by EPC at time $timestamp$ and 0 otherwise; and classify the tag as present if the reader detects it, and not present otherwise. A more sophisticated classifier may require additional features such as the past k readings of the tag before deciding on its location.

Definition 5.2.1 *A cleaning method is a classifier M that takes as input, a* tag case*, which is a tuple of the form $(\langle EPC, t\rangle, \langle f_1, f_2, ..., f_k\rangle)$, where each f_i is a feature describing one characteristic of the tag identified by EPC or its environment at time t, and makes a prediction of the form $(\langle EPC, t\rangle : loc, conf)$, where loc is the predicted*

location for the tag identified by EPC at time t, and $conf$ is the confidence that the classifier has in the prediction.

It is possible that some cleaning methods do not provide a confidence value with their predictions (in this case, we can consider that every prediction has a confidence of 100%). We will call such methods *terminal* methods, as the location they issue for each tag case is final, *i.e.*, at runtime we cannot call another cleaning method based on confidence.

We classify cleaning methods into three broad categories. The first category is user-defined rules that are provided by experts in the application domain and that can be used to quickly clean a large portion of an RFID data set. The second category is cleaning methods based on statistical models [13, 58, 29] that issue tag locations with a high probability of being correct. The third category is methods that use data mining techniques, such as a data warehouse, frequent patterns, or clustering to classify tag cases. For example, we can use historic information on flow patterns [40] or group movements [42] to determine the correct location of a tag. Multiple individual cleaning methods can be combined to form a new method. For example, a new method can be the combination of the window smoothing method, which detects false negatives, and pattern matching method, which detects false positives.

Cost model for a cleaning method. The cost of a cleaning method consists of the per-tuple cleaning cost and the error cost. The per-tuple cleaning cost is a function of two variables: 1) the amortized per tuple training cost and 2) the cost, in term of storage space and running time, for labeling a tag reading. The error cost is what we have to pay for each misclassified tag reading. The error cost can be a scalar value that simply penalizes issuing an incorrect tag location by a constant or a matrix[1].

Features available to a cleaning method. The context in which tag readings take place defines the feature space. Intuitively, features are important information regarding a tag at at the time of a reading. Features can be classified into four groups: (1) *Tag features*, which describe characteristics of the tag, such as communications protocol, vendor, price, or history of recent tag detections, (2) *Reader features*, which describe the reader, including the number of antennas, protocol, price, and vendor, (3) *Location features*, which describe the location where the reading took place, including the type of area being monitored (*e.g.*, door, shelf, and conveyor belt), or the sources of interference in the area, and (4) *Item features*, that describe the item to which the tag is attached, including item composition (*e.g.*, water or metal content), physical dimensions, or if it is a container.

A DBN-based view of cleaning

In this section we propose a new cleaning method based on Dynamic Bayesian Networks (DBNs) [79]. We assume there is a hidden process that determines the true location of a tag, but we only see noisy observations of the hidden

[1] In a more general setting the error may be a function of the distance of the correct location to the predicted location, the price of the item, and any other relevant features.

process. The model maintains a *belief state* that is the probability that the tag is present or not at a reader given the past observations. DBN-based cleaning has the advantage that it does not require us to remember recent tag readings, and as opposed to window smoothing, it gives more weight to recent readings (even within a window) than older ones.

A simple implementation of the model is to define a single hidden variable X_t that is true if the tag is present at the reader's location at time t, and a single observation variable e_t, which is a noisy signal dependent on X_t. We can compute the most likely current state X_{t+1} given the observations $e_{1:t+1}$ as:

$$P(X_{t+1}|e_{1:t+1}) \propto P(e_{t+1}|X_{t+1}) \sum_{x_t} P(X_{t+1}|x_t) P(x_t|e_{1:t}) \tag{5.1}$$

where $P(e_{t+1}|X_{t+1})$ is the known probability of observing a certain reading given a true state of the world, $P(X_{t+1}|x_t)$ is the known probability of changing from one true state to another, and $P(x_t|e_{1:t})$ is our previous belief state. The observation and transition models can be learned from the data or be given by the user. For example, we can update our observation model with the average detection rate of recent tag readings. The belief state is recomputed sequentially as we receive new readings. Figure 5.2 presents the graphical representation of the DBN.

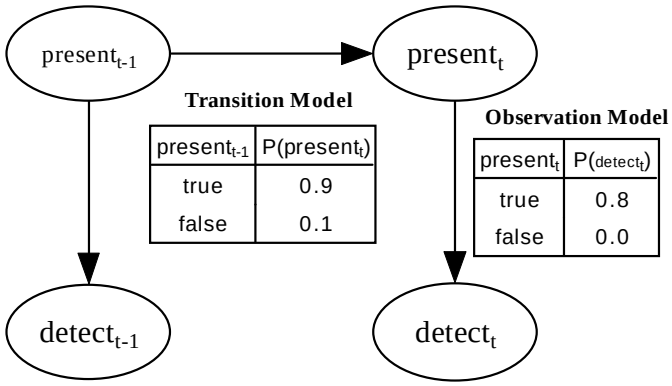


Figure 5.2: Structure of a simple DBN for cleaning

5.2.2 Cleaning Plan

A *cleaning plan* specifies the the conditions under which we should apply each cleaning method in order to minimize the expected total cost of cleaning a data set. A natural way to model a cleaning plan is to use a decision tree induced on a labeled set of tag cases. Each node in the tree can be split along the distinct values of a feature. The tag cases in each node share the same values on the features used to split the nodes from the root of the tree to the node. Leaves in the tree represent subsets of data that will be labeled using the same data cleaning strategy.

Figure 5.3 presents a cleaning plan induced on the label data set from Table 5.1. This plan divides the data set into three groups, each with a different cleaning strategy.

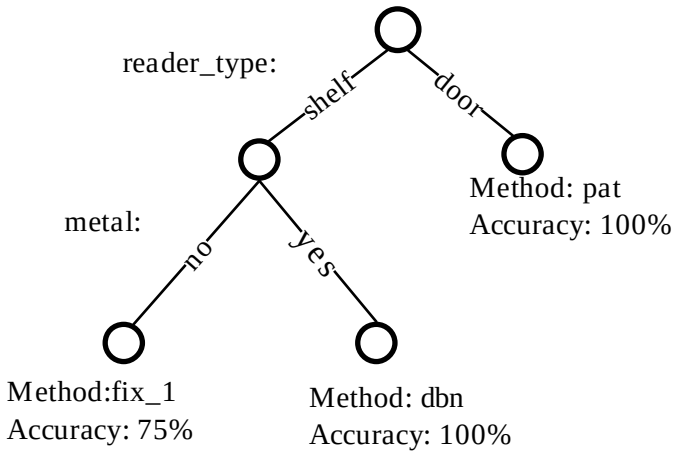


Figure 5.3: Example cleaning plan

Node Cleaning Costs

In order to decide how to construct the cleaning plan, we need a way to evaluate the expected cost to clean the tag cases residing at a node. This requires us to determine the optimal sequence of cleaning methods that should be applied to the cases in the node such that total cleaning cost is minimal.

A *cleaning sequence* is an ordered application of cleaning methods to a data set. The cost of applying a *cleaning sequence* $S_{D,M} = M_{s_1} \rightarrow M_{s_2} \rightarrow ... \rightarrow M_{s_k}$ to a data set D, denoted as $C(S_{D,M})$, is the sum of the cost of applying M_{s_1} to the complete data set D, applying M_{s_2} to the cases that M_{s_1} classified incorrectly, applying M_{s_3} to the cases that both M_{s_1} and M_{s_2} classified incorrectly, and so on, until applying M_{s_k} to the cases that all the other methods failed to classify, and the error cost incurred on the cases that no cleaning method is able to classify correctly. At cleaning, when applying a method to a case, we do not know if the label is correct. The only information available is the confidence that the method gives on the prediction. In practice, we can use confidence as threshold to determine if another method should be applied to the case.

Definition 5.2.2 ***Optimal cleaning sequence***. *Given a set of labeled tag cases D, and a set of cleaning methods $M = \{M_1, ..., M_k\}$, the* optimal cleaning sequence *to clean D with M, denoted as $S^*_{D,M}$, is the ordered sequence of methods from M with minimal cost.*

Determining the *optimal cleaning sequence* for a node, when we have non-terminal cleaning methods, is hard in practice, as methods can be correlated, *e.g.*, they may be good or bad at classifying the same cases, and confidences do not necessarily match classification correctness. In general, it is possible for high confidence predictions to provide wrong labels, and vice versa.

Optimal cleaning sequence approximation. We propose an approximation to the *optimal cleaning sequence* based

on the concept of *accuracy-adjusted cleaning cost*. This can be computed for terminal and non-terminal methods by taking each cleaning method as a black box that generates correctly classified cases at some cost. The cost is the per-tuple cleaning cost, adjusted by the percentage of mistakes that the method makes. A good strategy is to acquire as many correctly classified cases from low cost methods before we move to higher cost ones.

Given a non-terminal cleaning method M_i used to clean a data set D, the *accuracy-adjusted cleaning costs* is $C(M_i)/P_i(D)$, where $C(M_i)$ is the per-tuple classification cost of method M_i and $P_i(D)$ is the percentage of cases in D that M_i classifies correctly. If the method is terminal, this cost is computed as $C(M_i) + (1 - P_i(D)) \times E$, where E is the error cost. The reason for this distinction is that if we make a mistake with a terminal method, there is no way to be corrected later with another method.

A greedy approximation to the optimal cleaning sequence required to clean, data set D (residing at a leaf in the cleaning plan) can be computed iteratively as follows: (1) sort the cleaning methods on ascending order of *accuracy-adjusted cleaning costs*, and select the method with the lowest cost, (2) remove from D the tuples correctly classified by the method selected in Step 1, and (3) repeat the process until a terminal method is selected, or when the methods or cases are run out. Note that at each iteration of the greedy approximation we need to recompute the *accuracy-adjusted cleaning costs* of each method, as the data set requiring cleaning has changed. We will denote the greedy approximation to the optimal method sequence as $\hat{S}^*_{D,M}$.

There is one degenerate case: When all the methods are terminal, the optimal cleaning sequence can be obtained easily, by choosing the method with lowest *accuracy-adjusted cleaning cost*.

Example. Using the cost model from Example 2, the cost of the approximation to the optimal cleaning sequence for the root node of Figure 5.3, which is *dbn* →*pat*, can be computed as follows $2.0 + 0.33 \times 2.0 + 0.11 \times 5.0 = 3.21$.

Node Splitting Criteria

When deciding the structure of the cleaning plan, we need a criterion to determine when to split a node and the feature to choose for the split. This decision will be based on the difference in cleaning costs of the cases in the node before and after the split. We call this criteria *expected cost reduction*, defined as follows.

Definition 5.2.3 ***Expected cost reduction***. *Given a set of cleaning methods M and a data set D that can be split using feature f, with $|f|$ distinct values, into the subsets $D_1, \ldots, D_{|f|}$, we define the* expected cost reduction *of the split as,*

$$C(\hat{S}^*_{D,M}) - \sum_{i=1}^{|f|} \frac{|D_i|}{|D|} \times C(\hat{S}^*_{D_i,M}) \tag{5.2}$$

We should evaluate the *expected cost reduction* of each available feature and choose the one with the highest positive cost reduction. If no feature has positive cost reduction, or the cost reduction is below a certain *improvement*

Algorithm 3 Compute cleaning plan

Input: D: a labeled set of tag cases, $M = \{M_1, ..., M_k\}$: a set of cleaning methods with per-tuple cleaning costs $C(M_1), ..., C(M_k)$, E: the per tuple miss-classification cost.
Output: A cleaning plan P optimized for cost and accuracy
Method:

1: $q.push(D)$
2: **while** q not empty **do**
3: D' = $q.pop()$
4: $\hat{S}^*_{M,D}$ = compute cleaning cost of D'
5: **for** every available feature f **do**
6: split D' into $D_1, ..., D_{|f|}$ using f
7: C_f = compute weighted average of the cleaning cost of $D_1, ..., D_{|f|}$
8: **end for**
9: f^* = feature with minimum cost C_{f^*}
10: **if** $\hat{S}^*_{M,D} - C_{f^*} > 0$ **then**
11: push all subsets induced by f^* into q
12: **end if**
13: **end while**

threshold, we should not split the node. This threshold can be used as a means to reduce overfitting. Note that other overfitting prevention techniques are applicable to cleaning plan induction.

Example. If we again look at the cleaning plan in Figure 5.3, it has a cleaning cost of 3.21, if we split it on shelf, the new cleaning cost is the weighted average of 2.16 (shelf) and 2.0 (door), which is $2.16 \times 0.67 + 2.0 \times 0.33$ = 2.1. The benefit of this split is then $3.21 - 2.1 = 1.11$.

Cost of a Cleaning Plan

The cost of a cleaning plan is an important measure that indicates the expected per-tuple cleaning cost when using the plan to clean an RFID data set.

Definition 5.2.4 ***Cost of a cleaning plan***. *The total cost of applying a cleaning plan P to a data set D where each leaf in the plan $L_1, ..., L_m$ covers cases $D_1, ..., D_m$ is defined as follows:*

$$\sum_{i=1}^{m} \frac{|D_i|}{|D|} \times C(\hat{S}^*_{D_i,M}) \tag{5.3}$$

Example. The expected cleaning plan cost for the plan in Figure 5.3 is 2.11, which is $2.25 \times 0.44 + 2 \times 0.22 + 2 \times 0.34$. Comparing this value with the cost of classifying the whole data set without splitting which is 3.21, we have obtained a reduction of 35% in cost and have improved accuracy from 78% to 89%.

Labeled Data Acquisition

A critical component for the construction of the cleaning plan is the availability of labeled data. This data can be human labeled, which is an expensive option. A cheaper alternative is to use high-quality but expensive cleaning and sensing to get good data. For example, one can ask readers to interrogate a given tag in doubt by executing a *tag selection* operation. If the historic flow pattern for a given item is known, one can use this information to ask the set of most likely readers to inquire the particular tag in question and check its responses through *tag selection*. These techniques, although expensive and impractical for regular system operation, are valuable tools for obtaining highly reliable training data.

5.3 Algorithm: Cleaning Plan Induction

Algorithm 3 presents the algorithm for induction of the cleaning plan, based on a methodology of top-down induction of decision trees (TDIDT) [76]. Its main difference from a regular decision tree induction algorithm is at the node splitting criteria. We use the *expected cost benefit* based on the cost of the approximation to the *optimal cleaning sequence* for the parent node and its children. This is a greedy algorithm that finds good cleaning plans in practice, but it is not necessarily the optimal plan. The cost of a split is computed using the approximation method presented in the previous section.

5.4 Experimental Evaluation

In this section we present a thorough analysis of the performance of the *cleaning plan* on several data sets and compare its performance with the use of a number of cleaning methods applied individually. All the experiments were conducted on an AMD Athlon XP 1.2 GHz System with 1GB of RAM. The system ran cygwin 1.5.20-1 and gcc 3.4.4.

5.4.1 Data Synthesis

The data sets for our experiments were generated by a synthetic RFID data generator that simulates the operation of RFID readers under a wide variety of conditions. The simulation is composed of two components. The first simulates the movement of items through predefined paths. They are generated with a number of properties that alter the item behavior: (1) path speed, controlled by the duration (in *read cycles*) at each stage; (2) birth rate, the number of tags that enter the path at the beginning of each read cycle; (3) characteristics of items, such as water and metal content, which alter detection rates; and (4) item flow characteristics, where the paths can be traversed *sequentially* to simulate an item moving through the supply chain, *cyclically* to simulate a container traversing the same path repeatedly, and

randomly to simulate, *e.g.*, the movement of individuals. The second component simulates the operation of a number of RFID readers that detect the tagged items generated by the first component. The reader simulation is controlled by the following parameters: (1) reader and tag characteristics, which includes communication protocol, number of antennas, and price; (2) the RF noise characteristics of areas covered by the readers; and (3) the distance from a tag to each of the areas it covers. The simulation proceeds sequentially for a number of cycles. At each iteration, we simulate item movements and go through all the readers in the system simulating a read cycle, where the probability of tag detection is computed by compounding the effect of movement, item, tag, and reader characteristics.

Our experiments compare the performance of a cleaning plan, denoted as *plan* in the figures, with several cleaning methods used independently. Performance is measured in average per-tuple cleaning costs and accuracy, which is the percentage of correctly cleaned records. The first method is our proposed DBN-based cleaning, denoted as *DBN*. In our implementation, we used the structure presented in Figure 5.2, a fixed transition model, and an adaptable observation model that changes the probability of tag detection according to recent tag detection rates. The second method is the implementation of variable-window smoothing [58], denoted as *var*. The third method uses fixed-window smoothing, where three window sizes, 1, 2, and 3, are chosen and denoted as *fix_x*. We also implemented two user-defined rules, one denoted as *maj*, a combination of fixed window smoothing and conflict resolution by means of highest detection rate assignment, and the other is *pat*, for filtering false positive readings at doors or conveyor belts, by reporting as present only tags that have a detection rate that first increases as the item moves closer to the reader and then decreases as it moves away. In all the experiments, we induce the *cleaning plan* on a subset of the data (training set) and test the methods on the rest of the data (testing set). We set the training set to 30% of the complete data set. Data sets are generated by running the data simulator for 300 cycles.

The cost model used to evaluate the performance of the different cleaning techniques is as follows. The per tuple classification cost for each method was *DBN* 2.0, *var* 2.0, *fix_1* 1.0, *fix_2* 1.2, *fix_3* 1.4, *fix_4* 1.6, *pat* 2.5, and *maj* 1.4. These values were determined by considering the memory and time requirements of each method, and through experimental evaluation we verified that the values make sense. The only exception to this rule was to assign the same cost to *DBN* and *var* (although the former has lower costs) in order to simplify accuracy comparisons. The error cost we used was constant at 10 units. All the methods were defined as *terminal* except for *DBN* where we use the probability of a tag being present as confidence, and *var* where we use the distance to last tag detection in the window as confidence.

5.4.2 Cleaning Performance for a Diverse Reader/Tag Setup

In this experiment we compare the performance of the cleaning methods when applied to a data set obtained from a diverse configuration of readers and tags. This configuration is a small sample of the reader setups that may be

observed at a large RFID implementation. We use three readers that operate in low noise conditions, one reader has low detection rates caused by large distances to tags, two readers with variable detections rates due to distance to tags and the presence of metal in the area, one reader located at a conveyor belt, it should only detect tags on the belt, but that sometimes detects tags that go near the belt, and finally, three readers that have to detect tags attached to items that contain water or metal and thus have low detection rates. Some areas are covered by more than one reader, and conflicts are resolved by assigning tags to the closest reader.

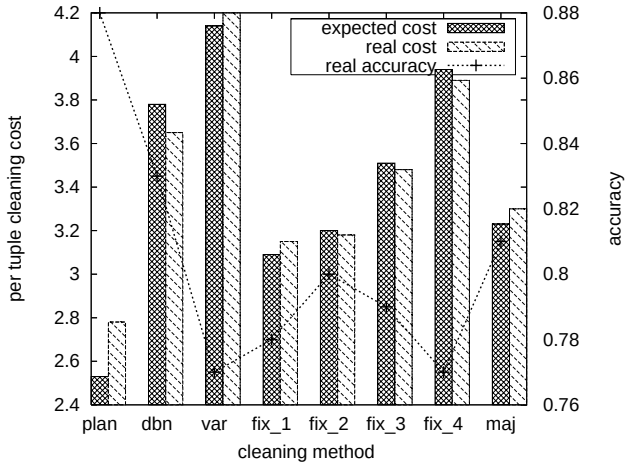


Figure 5.4: Cleaning performance for a complex setup

Figure 5.4 presents the expected (from training data) and the actual (from testing data) per tuple classification costs, and the accuracy of each method. As can be seen, *plan* is significantly cheaper and more accurate than any single method applied independently. The reason for such improvement is that the cleaning plan devised for this scenario correctly matches cleaning methods to the situations where they work better. It uses *fix_1* for the easy cases, *maj* for the cases when more than one reader detects a tag, *pat* for the conveyor belt reader, and multiple combinations of the methods, for the cases where RF noise is present.

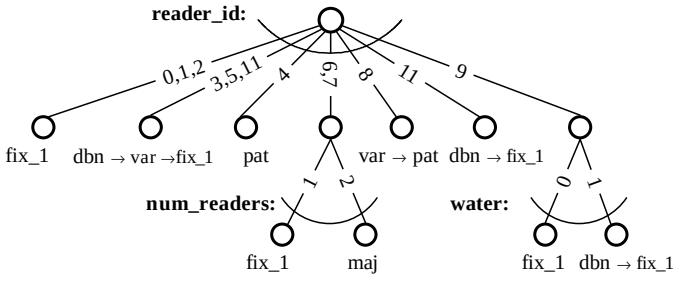


Figure 5.5: cleaning plan for complex setup

Figure 5.5 presents the *cleaning plan* induced for the complex setup. It uses *fix_1* for the easy cases, *maj* for the

cases when more than one reader detects a tag, *pat* for the conveyor belt reader, and multiple combinations of the methods, for the cases where RF noise is present. This plan makes intuitive sense as it matches methods to conditions correctly.

5.4.3 Cleaning Plan Performance for a Changing Reader/Tag setup

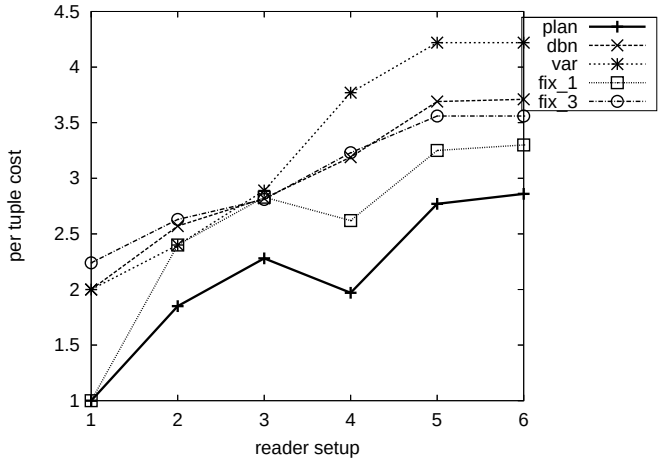


Figure 5.6: Cleaning costs vs. Setup complexity

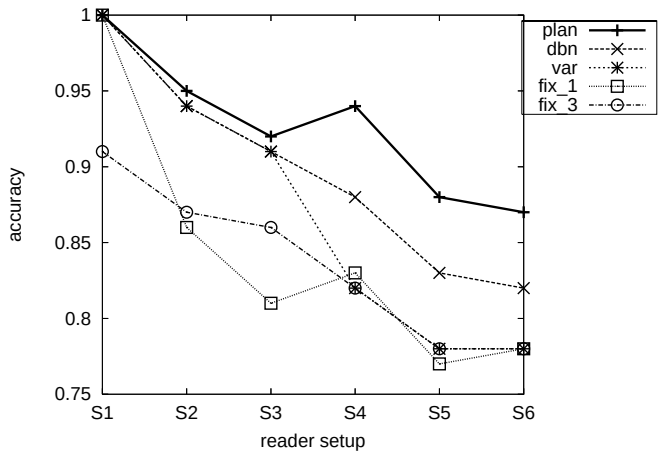


Figure 5.7: Cleaning Accuracy vs. Setup complexity

In this experiment we analyze the way the cleaning plan adapts to changing RFID configurations to outperform any individual cleaning method. This experiment tested six configurations, corresponding to adding one by one all the conditions present in the previous experiment. We start with easy cases with no noise and high detection rates, and increase the diversity up to the point that ends up with a complex configuration. From Figure 5.6 and 5.7, we can

see that in the first configuration, all the methods have low cost and 100% accuracy, and that *fix_1* performs better and cheaper than other techniques. (*Plan* in this case chooses *fix_1* as the only method to use). As we add new conditions, we see that the relative performance of *dbn* increases w.r.t. other individual methods, but *plan* adapts and has the best performance. This experiments highlights the difficulty in choosing a single method for a diverse RFID setup because methods work well under some conditions, but may fail when those conditions change.

5.4.4 Cleaning Plan Performance for Different Noise Levels

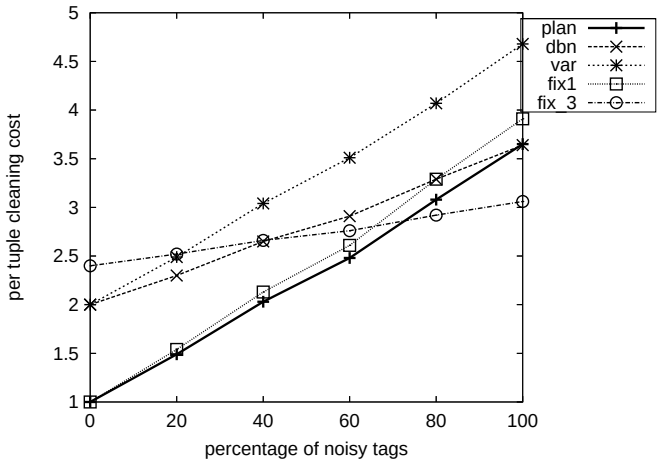


Figure 5.8: Cleaning costs vs. Percentage of noisy tags

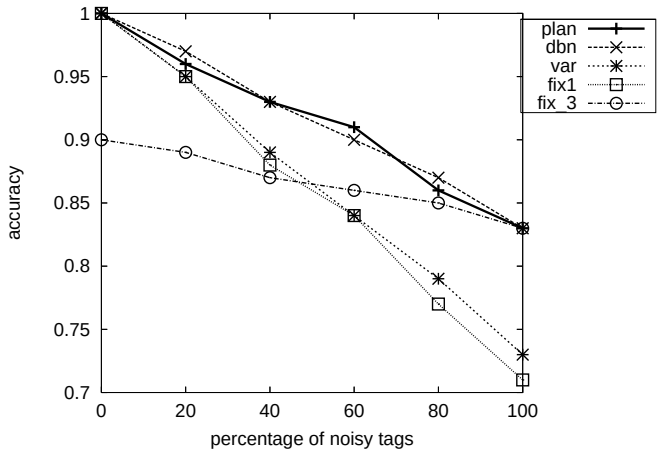


Figure 5.9: Cleaning Accuracy vs. Percentage of noisy tags

In this experiment we use a single reader that has to detect tags where a portion of them are attached to the items that cause significant RF interference, such as water or metal. This setup is typical of a retail store where a percentage

of the tags are attached to items that cause interference. From Figures 5.8 and 5.9, we observe that *plan* is always the best method or close to the best. By looking at the cleaning plan for each case, we see that the induction algorithm is dividing the tag population in two subgroups. The first is the case where the tags have little interference (using a feature that describes the items to which tags are attached) and can be cleaned with a cheap method, such as *fix_1*. The second is a more difficult case where a method with a larger smoothing window is chosen. The overall cost is thus reduced by avoiding using expensive techniques on easy cases.

5.4.5 Cleaning Method Availability

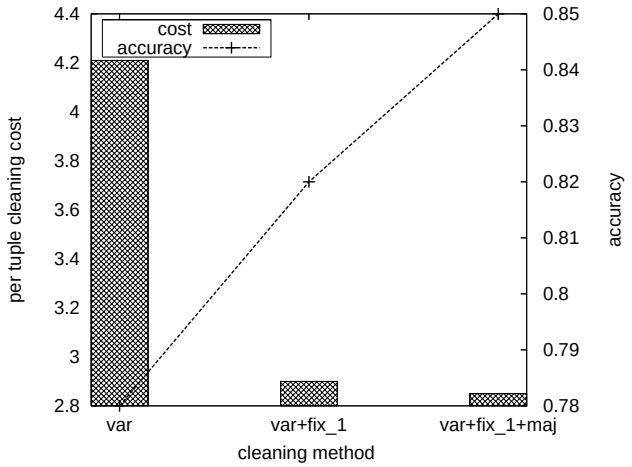


Figure 5.10: Cleaning performance vs. Available cleaning methods

In this experiment we change the number of cleaning methods available for the construction the cleaning plan. We conducted this experiment on the diverse reader setup data set. In Figure 5.10 we compare three scenarios, the first when only the method *var* is available, the second where *var* and *fix_1* are available, and the third when *var*,*fix_1*, and *maj* are available. The results show the importance of using a combination of cleaning techniques to improve accuracy and reduce cost. We see that the cleaning plan learns when to apply each method effectively and in doing so gains a significant reduction in cost with gains in accuracy.

5.4.6 Time to Induce the Cleaning Plan

In this experiment we evaluate the runtime of the cleaning plan induction algorithm for different training data set sizes, and for different number of cleaning methods available. This experiment was conducted using the same setup of the first experiment. As can be seen from Figure 5.11 the time increases linearly with the number of training cases. Larger number of available methods increases the slope of the runtime curve, this is because each iteration of the cleaning plan

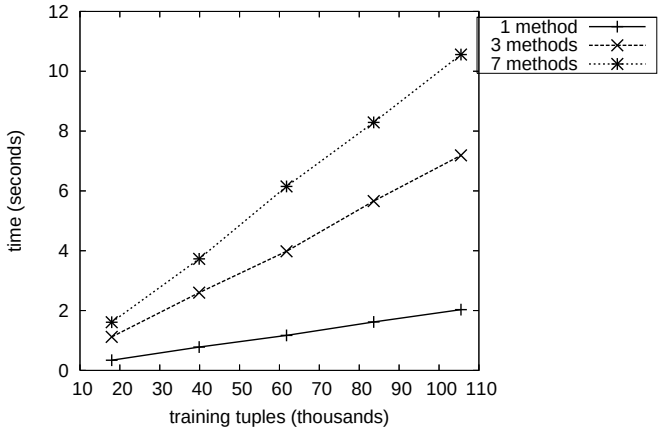


Figure 5.11: Runtime vs. Training data set size

induction algorithm needs to consider more complex cleaning sequences. We avoid an exponential runtime increase by using our greedy approximation to the *optimal cleaning sequence* and the *cleaning plan*.

5.4.7 Tag Speed

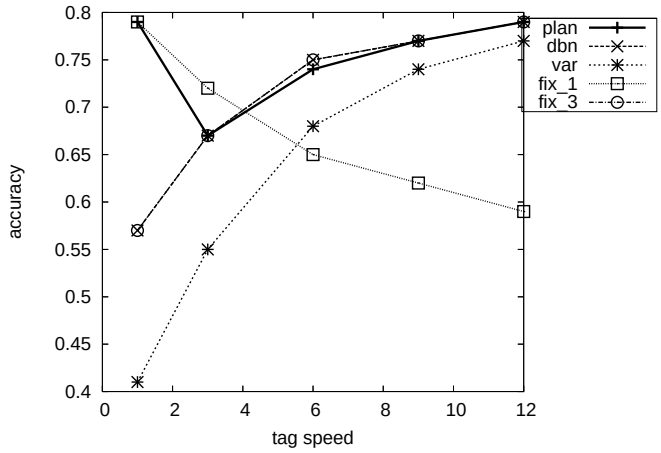


Figure 5.12: Tag Speed vs. Accuracy

In this experiment, we examine the difficult problem of detecting rapidly moving tags. In this experiment, speed is measured as the average number of cycles that tags remain at each location before moving. From Figure 5.12 , we observe an interesting pattern: At very high speed (where item remains for only one cycle at the location) *fix_1* works better (as expected since items remain for at most one cycle at a location), and *plan* chooses this method. As the speed slows down, the performance of *fix_1* decreases, and the cleaning plan uses *dbn* that has good accuracy for

lower speeds. For this experiment, the cleaning plan suffers some penalty on cost (not shown in the figure) because it uses *dbn* which has higher costs than the fixed-window methods of different sizes.

5.4.8 Cost of Errors

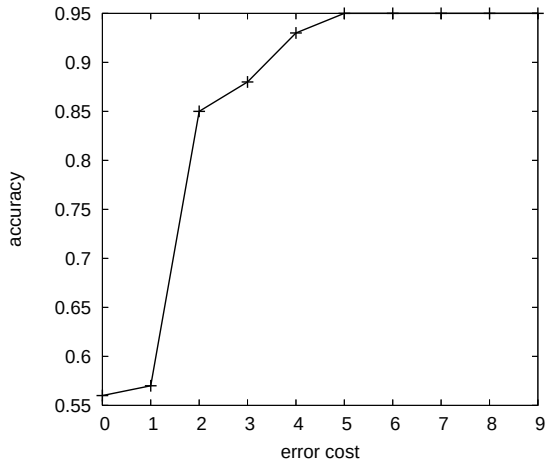


Figure 5.13: Error Cost vs. Accuracy

This experiment shows the effect that changing the error cost has on the accuracy of the cleaning plan. In Figure 5.13 we see that when we assign a cost of 0 units to an error, the plan induction algorithm decides that the optimal solution is to "guess" tag locations and thus accuracy is only around 50%. As we increase the error cost the induction algorithm favors the utilization of more expensive but more accurate plans, up to a point where the plan with best accuracy has been selected and higher error costs have little effect on plan structure. This analysis is useful in determining a good error cost for inducing the cleaning plan, situations were very high volumes of information need to be cleaned quickly, may allow for smaller error costs and faster plans, but if accuracy is a must error cost should be high.

5.4.9 Cleaning Plan Induction

The process of inducing a cleaning plan is quite efficient. From figure 5.14 we see that the time to induce the cleaning plan grows only linearly with the size of the training data set. Cost growth with respect to the number of features is much steeper (not shown in the figure), because each node of plan requires the evaluation of many more tuples, and thus more scans of the training data. Our implementation of plan learning uses a basic top down decision tree induction algorithm that repeatedly scans the data at each iteration, in practice we can use an algorithm based Rainforest [38] to further reduce the costs when large training data sets with large numbers of features are used.

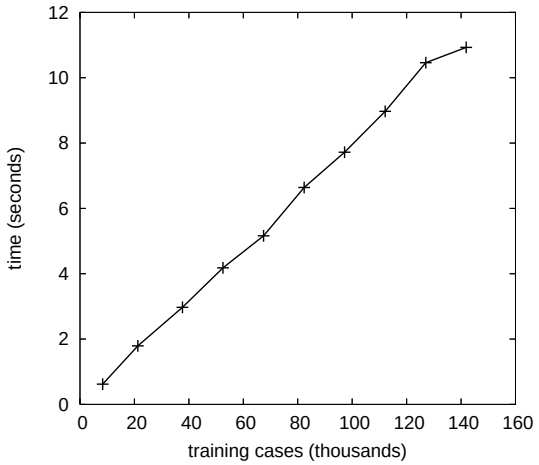


Figure 5.14: Training Cases vs. Time

5.5 Summary

In this chapter we have proposed a novel cost-conscious framework for cleaning RFID data sets that learns when and how to apply different cleaning techniques in order to optimize total cleaning costs and accuracy. The key intuition behind the framework is to determine the situations in which inexpensive cleaning methods work well, while only applying expensive methods when really necessary.

An efficient method has been proposed for cleaning plan construction, which adopts the idea of top-down induction of decision trees and applies a novel feature selection criterion based on the concept of *cost reduction* that splits the RFID data set along the feature values that provide the highest expected reduction in cleaning costs. The cleaning plan can be constructed at training time and significantly improve cleaning speed and accuracy at runtime. Moreover, we proposed a DBN-based cleaning method, which takes tag readings as noisy observations of a hidden state and performs effective data cleaning.

Our empirical study verifies the ability of the cleaning plan to be adapted to diverse RFID configurations and outperform all existing methods that are applied individually in cost and accuracy. It also shows that our cleaning plan can be efficiently computed, and that the proposed algorithm generates plans that are efficient and make intuitive sense.

Chapter 6

FlowGraph Mining

An important application of moving object analysis is mining movement patterns of objects in supply chain operations. In this context, one may ask questions regarding correlations between time spent at quality control locations and laptop return rates, salient characteristics of dairy products discarded from stores, or ships that spent abnormally long at intermediate ports before arrival. The gigantic size of such data, and the diversity of queries over flow patterns pose great challenges to traditional workflow induction and analysis technologies since processing may involve retrieval and reasoning over a large number of inter-related tuples through different stages of object movements. Creating a complete workflow that records all possible commodity movements and that incorporates time will be prohibitively expensive since there can be billions of different location and time combinations. We propose the FlowGraph [41], as a compressed probabilistic workflow, that captures the general flow trends and significant exceptions of a data set. The FlowGraph achieves compression by recording the set of major flow trends, and the set of non-redundant flow exceptions (*i.e.*, abnormal transitions or durations) present in the data.

6.1 Clustered Path Database

For RFID data flow analysis, one is in general, interested only in the *duration* (*i.e.*, length of time) of object stay or transition, but not the *absolute time*. In such cases, we can an RFID dataset as a set of paths of the form $(EPC : (l_1, d_1)\ (l_2, d_2) \ldots (l_k, d_k))$, were l_i is the i-th location in the path traveled by the item identified by EPC, and d_i is the total time that the item stayed at l_i.

The duration that an item spends at a location is a continuous attribute. In order to simplify the model even further we can discretize all the distinct durations for each location into a fixed number of clusters. We call such a path-database with discretized durations *clustered path-database*. Table 6.1 presents an example of a clustered path-database, where, (l_j, i) corresponds to an item that stays for the duration i (where i is a cluster) at location l_j.

$Path$	$count$
$(l_1, 1)$	700
$(l_1, 2)$	700
$(l_1, 3)$	2400
$(l_1, 1) \rightarrow (l_2, 1)$	240
$(l_1, 2) \rightarrow (l_2, 1)$	240
$(l_1, 3) \rightarrow (l_2, 1)$	800
$(l_1, 3) \rightarrow (l_3, 1)$	2160
$(l_1, 3) \rightarrow (l_3, 2)$	2160
$(l_1, 3) \rightarrow (l_3, 1) \rightarrow (l_1, 1)$	18
$(l_1, 3) \rightarrow (l_3, 1) \rightarrow (l_1, 2)$	18
$(l_1, 3) \rightarrow (l_3, 1) \rightarrow (l_1, 3)$	144
$(l_1, 3) \rightarrow (l_3, 1) \rightarrow (l_1, 1) \rightarrow (l_2, 1)$	6
$(l_1, 3) \rightarrow (l_3, 1) \rightarrow (l_1, 2) \rightarrow (l_2, 1)$	6
$(l_1, 3) \rightarrow (l_3, 1) \rightarrow (l_1, 3) \rightarrow (l_2, 1)$	48
$(l_1, 3) \rightarrow (l_3, 1) \rightarrow (l_1, 1) \rightarrow (l_3, 1)$	36
$(l_1, 3) \rightarrow (l_3, 1) \rightarrow (l_1, 2) \rightarrow (l_3, 1)$	36
$(l_1, 3) \rightarrow (l_3, 1) \rightarrow (l_1, 3) \rightarrow (l_3, 1)$	144
$(l_1, 3) \rightarrow (l_3, 1) \rightarrow (l_1, 3) \rightarrow (l_3, 2)$	144

Table 6.1: A clustered path-database

6.2 RFID Workflow

A clustered path-database can be naturally modeled as a probabilistic workflow. Each location corresponds to an activity, and locations are linked according to their order of occurrence. Links between activities q and p have an associated probability that is the percentage of the time that activity p occurred right after activity q. We will illustrate the benefit of such modeling with an example: The US government will require every container arriving at ports in the country to carry an RFID tag in order to detect abnormal paths. We can easily determine the level of abnormality of a path by looking at its transition probabilities in the worflow; you could even look at the current trajectory of a container and predict the set of the most likely next stages, and raise a real time alert when an in-transit container has taken an unexpected transition.

We will state a more formal definition of a probabilistic workflow that is based on the definition of a probabilistic deterministic finite automaton *PDFA* [56]. The only difference is that we associate states with symbols of the alphabet whereas the *PDFA* associates transitions with symbols.

A probabilistic workflow is defined by the tuple
$(Q, \Sigma, \eta, \tau, q_0, F)$ where

- Q is a finite set of states
- Σ is the alphabet, in our case it will be locations
- $\eta : Q \rightarrow \Sigma$ is the naming function, that assigns a symbol from Σ to each state
- $\tau : Q \times Q \rightarrow [0, 1]$ is a function that returns the probability associated with a transition

- q_0 is the initial state,
- $F : Q \rightarrow [0, 1]$ is a termination function which returns the probability for a state to be final.

The sum of all the transition probabilities for a state plus its termination probability should add up to 1. Figure 6.1 presents a probabilistic workflow constructed on all the paths from Table 6.1. The transition function value for each pair of states $l_i \rightarrow l_j$ is the number placed on the arrow, and it is computed as $count(l_i, l_j)/count(l_i)$, where $count(l_i, l_j)$ is the number of items that took the transition, and $count(l_i)$ is the total number of items that arrived at l_i. The termination function value for each node l_i is the number placed on top of the node and is computed as $count(l_i, \#)/count(l_i)$, where $count(l_i, \#)$ is the number of items that terminate their path at l_i. This workflow is a highly compact summary of the data; we have represented all the paths in the database with a model that has just 6 nodes. But the compression is not lossless, the workflow in Figure 6.1 does not have any information on the duration spent at each location. For example, we cannot distinguish the paths $(l_1, 1)(l_2, 1)$ and $(l_1, 2)(l_2, 1)$ of Table 6.1.

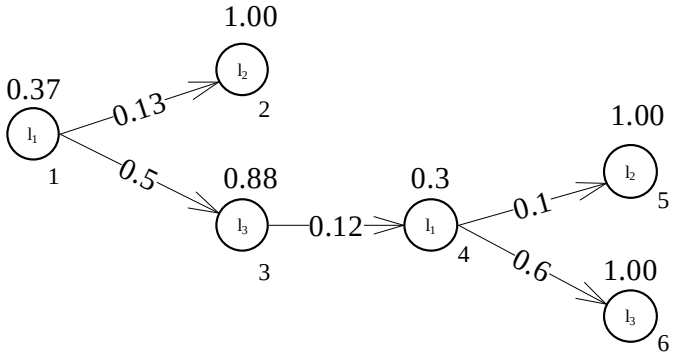


Figure 6.1: Probabilistic workflow

6.2.1 Complete Workflow

Creating a workflow that incorporates duration information is valuable in the analysis of RFID paths. Duration may for example be very important in inventory management; we can detect path segments that have unusually long durations and use the information to re-route items if possible, or to optimize those segments. In order to incorporate duration into the workflow, the first option is to extend Σ to be not just locations, but the cartesian product of locations $\times$ durations. We refer to a workflow with such nodes as a *complete workflow*. Figure 6.2 presents the *complete workflow* for Table 6.1. The problem with this approach is that in many cases we will get a very significant increase in the number of nodes (in our example we went from 6 nodes to 19), with a large percentage of the new nodes being uninteresting. For our running example, we would replace the first node l_1 with 3 nodes $(l_1, 1)$, $(l_1, 2)$, and $(l_1, 3)$, l_2 would have only one node $(l_2, 1)$; if we look at the transition $(l_1, 3) \rightarrow (l_2, 1)$ the transition probability is 0.1 a number

close to 0.13 the value recorded in the duration independent workflow, for some applications, explicitly identifying this transition does not add much information. Redundancy becomes more serious when examining all the possible transitions between two nodes with many different possible durations each but with a duration distribution largely independent of each other.

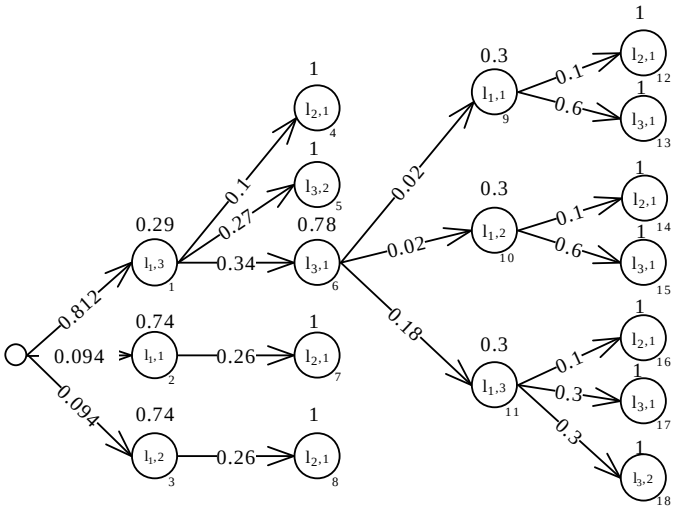


Figure 6.2: Complete workflow

6.2.2 Extended Probabilistic Workflow

In order to solve the inefficiencies presented in the *complete workflow* model we develop an *extended probabilistic workflow* model that incorporates durations by factoring probabilities into conditional transition probabilities, associated with transitions, and conditional emission probabilities associated with durations at nodes. This model is similar to a Hidden Markov Model (HMM) [78] but in our case, we conditioned the emission and transition probabilities at a state on the durations at all previous states, a concept not present in HMMs.

More formally, an *extended probabilistic workflow model* is a tuple $(Q, C, \Sigma, \eta, T, E, q_0)$, where all the components are defined as in a *probabilistic workflow*, and C, T, and E are defined as:

- C, is the set of possible durations at any node, including * to represent any duration.
- $T : \mathcal{P}owerset(Q \times C) \times (Q \bigcup \{\#\}) \rightarrow [0, 1]$, a conditional transition probability function that assigns a probability to each transition going into a state Q or $\#$ (the special termination state symbol) given the time spent at each of the predecessors of Q.
- $E : \mathcal{P}owerset(Q \times C) \times (Q \times C) \rightarrow [0, 1]$, a conditional emission probability function that assigns a probability to each length of stay $c \in C$ at node Q given the time spend at each of the predecessors of Q.

As before, we require that the emission and transition probabilities conditioned on a given path prefix add up to 1.

The ability to compute the probability of a path or a path segment is important, it can be used for example for data cleaning, we can use the workflow to detect unlikely transitions due to false positive readings, or to check the most likely next locations where an item can go right after being at a given location, in order to detect if the reader missed the item or if it really moved. The workflow can also be used to rank paths according to likelihood and allow data analysts to focus their effort in optimizing the top-k most likely paths.

Before we present how to compute probabilities in an *extended probabilistic workflow* we need a few definitions:

- We define a path x to be the sequence of stages $x_1\ x_2 \ldots x_l$, where each x_i a location/duration pair of the form (q, c); $q \in Q$ and $c \in C$. When $c = *$ when we do not care about the particular duration at the location.
- The length of a path x, $length(x)$ is the number of stages in the path.
- We can identify the state and duration of components of the stage i in path x by the functions $state(x, i)$ and $time(x, i)$ respectively.
- We define the prefix of a path $prefix(x, i)$ to be the first i stages in path x. $prefix(x, 0) = \phi$. The prefix of a path, is a path itself. Every path is a prefix of itself.
- The function $count(x)$ returns the number of times that the path x appears in the clustered path database as a complete path or as a prefix of a longer path x'. Note that $(q, c = *)$ matches any duration at location q.
- A path xx' is the concatenation of the stages of the paths x and x'. A path $x(q, c)$ is the concatenation of the stages of path x with the stage (q, c).

We can define the probability of a path as the product of all the conditional emission probabilities for the durations spent at each location in the path, multiplied by the product of all the conditional transition probabilities in the path.

$$P(x) = Emission(x) \times Transition(x) \tag{6.1}$$

$$Emission(x) = \prod_{i=1}^{n} E(prefix(x, i-1), x_i) \tag{6.2}$$

$$Transition(x) = \left(\prod_{i=1}^{n} T(prefix(x, i-1), state(x_i)) \right) \times T(x, \#) \tag{6.3}$$

The computation of an emission probability for c_i ($c_i \neq *$) in the stage $x_i = (q_i, c_i)$ given the prefix $x_1 \ldots x_{i-1}$ is done as follows:

$$E(prefix(x, i-1), x_i) = \frac{count(prefix(x, i))}{count(prefix(x, i-1)(state(x_i), *))} \tag{6.4}$$

The computation of a transition probability to location l_i given the prefix $x_1 \dots x_{i-1}$ is done as follows:

$$T(prefix(x, i-1), l_i)) = \frac{count(prefix(x, i-1)(l_i, *))}{count(prefix(x, i-1))} \tag{6.5}$$

Example (Emission probability computation). Figure 6.3 presents each node in the workflow with the conditional emission and transition probabilities. Rows represent durations at the node, and columns the probability of the duration conditioned on different path segments. If we look at the conditional emission table for node 6, we can compute cell 1 (row 1, column 1) as $E((l_1, *)(l_3, *)(l_1, *), (l_3, 1)) = count((l_1, *)(l_3, *)(l_1, *)(l_3, 1))/count($ $(l_1, *)(l_3, *)(l_1, *)(l_3, *)) = 216/360 = 0.6$. Similarly we can compute transition probabilities.

Uninteresting Conditional Probabilities

When computing the conditional transition and emission probabilities for a node, we do not need to consider every possible path that is a prefix to the node, as a conditioning factor. In certain supply chain applications it is possible for some items to have unique path segments, *e.g.*, items moving in a conveyor belt may all have identical paths and durations at each stage. When finding the conditional emission or transition probabilities we can consider each unique path segment as an atomic unit, this offers great benefits in computation costs and workflow compression, *e.g.*, if every item of a certain type goes through the same 20 stages in a conveyor belt, by considering that path segment as a unit we gain a 2^{20} to 1 reduction in the number of possible path prefixes to consider.

We call a conditional emission or transition entry *uninteresting* if it can be derived from another entry without loss of information. For example, if we know that before a certain product gets to the shelf, it always spends 1 day in the warehouse, 2 days in the store backroom, and 4 hours in the truck, there is no need to condition the length of stay at the shelf on all eight combinations of these three stages, conditioning on the single prefix containing all stages is sufficient.

We say that a path x is closed iif you can not find another path y that goes thorough the same stages as x and that is more specific (has less durations set to *) and $count(x) = count(y)$. This is a concept very similar to that of closed frequent patterns [72]. An emission or transition probability entry conditioned on x is *uninteresting* if x is not closed.

Lemma 1. Conditional emission probabilities conditioned on non-closed paths can be derived without loss of information from conditional emission probabilities conditioned on closed paths.

Proof Sketch: Assume that we are looking at node q_i of a workflow and computing the emission probability of duration c_i. Assume that x is a non-closed path leading to q_i and that y is the closed path corresponding to x. If we know $E(y, (q_i, c_i))$, given that $count(x) = count(y)$ we get that $E(x, (q_i, c_i)) = E(y, (q_i, c_i))$. ∎

Lemma 2. Conditional transition probabilities conditioned on non-closed paths can be derived without loss of information from conditional emission probabilities conditioned on closed path.

Proof Sketch: Using an analogous argument to that used in the proof of lemma 1, the transition probability $T(x, q_i) = T(y, q_i)$ when x is not closed and y is the corresponding closed path. ∎

Example (Uninteresting conditional probabilities). Looking up node 3 in the workflow of Figure 6.3 and computing the conditional transition probability to node 4 given $(l_3, 1)$, we get $T((l_1, *)(l_3, 1), l_1) = 600/2760$. This is exactly the same as $T((l_1, 3)(l_3, 1), l_1) = 600/2760$. The reason is that $(l_1, *)(l_3, 1)$ is not a closed path. Thus the first conditional probability is uninteresting.

Redundant Conditional Probabilities

The benefit of all conditional emission and transition probabilities is not the same. Intuitively, a conditional probability that deviates significantly from the probability that we would infer from the currently recorded conditional probabilities is more important than the one that deviates very little. Such deviations are important because they are interesting patterns by themselves, *e.g.*, it may be important to discover that the transition, for perishable products, to the return counter increases when they spent too long in transportation locations. Recording these deviations is also important to compute more accurate path probabilities.

When determining if a conditional probability is redundant at a node it is important to *condition on short path prefixes before we condition on long path prefixes*, *i.e.*, we only record deviations given prefixes of length i if they are not redundant given recorded deviations on prefixes of length $i - 1$. This optimization is quite natural and effective as it will uncover substantially fewer truly "surprising" deviations, that are really interesting.

Let us define $\hat{E}(x, (q, c))$ to be the expected emission probability of duration c at node q given path x. If we have already computed the emission probabilities $E(a_1, (q, c)), \ldots, E(a_n, (q, c))$, where each a_i is a direct ancestor of x (same path stages, and just one less duration set to *) and $(a_i, (q, c))$ and $(a_j, (q, c))$ are independent and conditionally independent given (q, c) for every $i \neq j$, we can compute $\hat{E}(x, (q, c))$ as,

$$\hat{E}(x, (q, c)) = \left(\prod_{i=1}^{n} \frac{E(a_i, (q, c))}{E(b, (q, c))} \right) \times E(b, (q, c)) \tag{6.6}$$

where $E(b, (q, c))$ is the unconditional probability of observing duration c at state q, b is the path up to q with all durations set to $*$. Intuitively, this equation is adjusting the unconditional probability of observing duration c in stage q, by compounding the effect that each ancestor independently has on the emission probability[1].

[1] Eq. (6.6) can be better understood with a simple example, assume that you have three events a, b, and c, and b, c independent and conditionally independent given a, you can compute $P(a|bc) = P(a) \times P(a|b)/P(a) \times P(a|c)/P(a)$, which is equivalent to our formula when you have only two ancestors.

We say that a conditional emission probability is *redundant* if $|\hat{E}(x,(q,c)) - E(x,(q,c))| < \epsilon$. And we call ϵ the *minimum probability deviation threshold*[2].

Similarly, we can define *expected transition probability* of $T(x,q)$ as:

$$\hat{T}(x,q) = \left(\prod_{i=1}^{n} \frac{T(a_i,q)}{T(b,q)} \right) \times T(b,q) \tag{6.7}$$

where each a_i a direct ancestor of x and $T(b,q)$ is the unconditional transition probability to state q. The intuition behind Eq. (6.7) is the same as for Eq. (6.6).

We say that a conditional transition probability is *redundant* if $|\hat{T}(x,q) - T(x,q)| < \epsilon$.

Low Support Conditional Probabilities

Up to this point we have considered all the closed paths that are a prefix of a node in computing its conditional emission and transition probabilities. Conditioning probabilities on every closed path may lead to very large probability tables, with many entries supported by very few observations (paths). The size of our workflow may end up being dominated by noise in the clustered path-database. In order to solve this problem, we restrict our closed paths to only those that appear in the clustered database more than δ percent of the time. This has the effect that any probability entry will be supported by at least δ percent of the observations and the size of the tables will be significantly smaller. We call δ the minimum support.

Compressed Workflow - FlowGraph

We call an *extended probabilistic workflow*, with only interesting, and non-redundant conditional emission and transition probability entries, conditioned on closed paths with minimum support δ and minimum probability deviation ϵ a FlowGraph.

The optimal definition of δ and ϵ, such that a good compromise between workflow size, exception interestingness, and information loss is achieved is a very challenging task, and one that depends on the specific application needs. When the main objective is to detect general flow patterns and significant exceptions, the use of larger values of δ and ϵ has two advantages: discovered exception will likely be more interesting, and the total number of exceptions to analyze will be smaller. When the objective is to use the FlowGraph to compute path probabilities with high accuracy, we need to be more conservative in the selection of the parameters. For this case a good indicator for parameter selection may be the nature of the paths for which we will estimate probabilities. ϵ can be estimated by checking the error on a representative query set with varying levels of the parameter, we can use the value where error rate has a big jump (see Figure 6.7). δ could be selected using a top-k frequent path approach such as [51].

[2]Alternative definitions are possible. For example, we could define an emission probability to be redundant if: $max_{a_i}(|E(x,(q,c)) -$

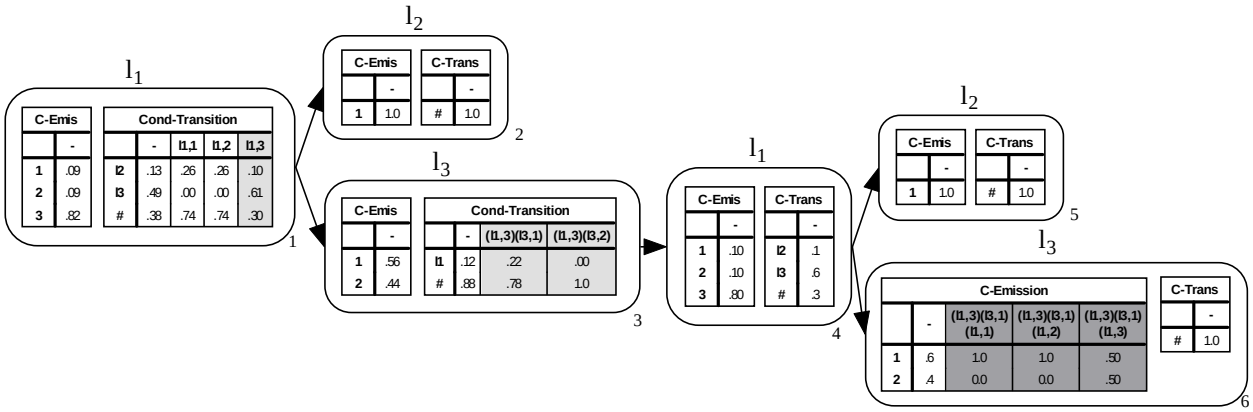


Figure 6.3: FlowGraph

Figure 6.3 shows the FlowGraph derived from the clustered path database in Table 6.1. The number of nodes is 6, the same as the duration independent workflow in Figure 6.1. Each node contains a conditional emission table, and a conditional transition table. The lightly shaded columns (nodes 1 and 3) are redundant for an $\epsilon = 0.2$ and the darkly shaded columns (node 6) are not supported for a $\delta = 150$ paths. An estimate for the size of the workflow is the number of probability entries, in this case, 30 entries [3].

6.3 FlowGraph Computation Method

In this section we will introduce an algorithm to compute a FlowGraph from a clustered path database, with minimum support δ and minimum probability deviation ϵ.

6.3.1 Mining Closed Paths

From the discussion in the previous section we know that we only need to condition transition and emission probabilities on frequent closed paths. The mining of closed paths can be easily implemented with an efficient closed itemset mining algorithm [91] after associating locations to nodes in a duration independent workflow . If we are computing not a single cell but a complete FlowCube we can compute all the frequent closed paths for all cells at all levels of abstraction in parallel using the method presented in [40].

After we have computed the counts of all closed paths, we need to do one more pass over the path database computing the support of each closed path followed by each duration independent suffix. For example, if we get that $(l_1, 3)(l_3, 1)$ is a frequent closed path, we should compute $(l_1, 3)(l_3, 1)(l_1, *)$, $(l_1, 3)(l_3, 1)(l_2, *)$, and $(l_1, 3)(l_3, 1)(l_3, *)$.

$E(a_i, (q, c))| < \epsilon$.

[3] In reality the number of probabilities is smaller as we only need to store $n - 1$ entries for each column given the law of total probability.

These counts will be needed in computing conditional emission and transition probabilities.

6.3.2 Mining Conditional Probabilities

When computing conditional probabilities given closed frequent paths we follow a short to long principle. We first record unconditional probabilities, then we record probabilities conditioned on paths of length 1 and that deviate from the unconditional probabilities; when we have all paths of length 1, we look at the ones of length 2, and record deviations given the unconditional entries and the length 1 entries; we repeat this process until we have looked at all closed frequent path prefixes for a node. This method is very efficient as the number of distinct short paths is quite small as compared to large paths, and in real RFID applications conditional probabilities usually depend on short path segments, *e.g.*, the transition to the location return counter may depend on the time that an item stayed at quality control, but not on a very long combination of locations and durations.

Computing Non-redundant Conditional Emission Probabilities

At this point we have all the information required to compute the conditional emission probabilities given by Eq. (6.4) for all nodes in the workflow.

When computing $E(prefix(x, i-1), x_i)$ we require the entire path x to be frequent and not only the $prefix(x, i-1)$ part to be frequent. This makes sense as we want to compute deviations in emission probability when enough supporting evidence (including the emission duration itself) is available.

We need to determine which conditional probabilities are redundant and that can be easily done by arranging the closed paths that are a prefix of each node in a lattice, and traversing the lattice from more general paths to more specific paths, storing only probabilities that are found to be non-redundant using Eq. (6.6).

Computing Non-redundant Conditional Transition Probabilities

We can use Eq. (6.5) to compute the conditional transition probabilities for all nodes in the workflow.

When computing $T(prefix(x, i-1), q_i)$ we require $prefix(\ x, i-1)\ (q_i, *)$ to be frequent. This guarantees that we have enough evidence of the conditional transition to q_i.

The determination of redundant conditional transition probabilities follows exactly the same logic used for emission probabilities in the previous section.

6.3.3 Algorithm

Based on the previous discussion we state algorithm 4, which summarizes the complete set of steps required to output the FlowGraph.

Algorithm 4 Construct FlowGraph

Input: Clustered path database D, minimum conditional probability deviation ϵ, minimum support δ
Output: FlowGraph W
Method:

1: On a single pass over D construct the duration independent workflow W_1.
2: Using W_1 compute D_1 the path database with locations encoded with nodes in W_1.
3: Call a frequent closed itemset mining algorithm with D_1 as the transaction database and minimum support δ to derive C the set of closed frequent paths.
4: Scan D once to collect the support of the extended frequent closed paths.
5: Construct W by annotating each node in W_1 with the set of non-redundant conditional emission and transition probabilities given the closed paths in C, and the minimum conditional probability deviation ϵ.
6: return W.

Analysis. Algorithm 4 can be executed quite efficiently, in addition to the scans required by the frequent closed itemset mining algorithm, we need just two more scans, one to build the duration independent workflow, and another to collect extra counts needed to compute conditional probability entries.

6.3.4 Computing the Probability of a Path in the FlowGraph

Eq. (6.1) can be used to compute the probability of a path x when we have all possible conditional emission and transition probabilities. But when we have compressed the workflow by using ϵ as a threshold for redundant probabilities, and δ as the minimum support for recording a conditional probability, we can not use this equation anymore, because many of the required probabilities may be missing.

The solution is to define an approximation of $P(x)$ called $\hat{P}(x)$, that uses Eqs. (6.6) and (6.7) for computing conditional emission and transition probabilities respectively, whenever the exact probabilities have not been explicitly recorded in the compressed model. The path prefixes to use are the most specific paths recorded in the model that are still more general than x.

The probability for path x computed by $\hat{P}(x)$ is an approximation to $P(x)$ and is subject to error. The amount of error will increase for larger values of ϵ and δ, and will also depend on the support of x. If $P(x)$, that is the true probability of x is very low, then the error can be larger as the computation of the probability of x may require conditional probability entries that were not stored in the FlowGraph because of not having enough support. In the experimental section we explore the conditions under which error increases in more detail.

6.4 Experimental Evaluation

In this section, we perform a thorough analysis of our model and algorithms. All experiments were implemented using C++ and were conducted on an Intel Pentium IV 2.4GHz System with 1GB of RAM. The system ran Debian Sarge with the Linux kernel 2.6.13.4 and gcc 4.0.2.

6.4.1 Data Synthesis

The RFID databases in our experiments were generated using a random path generator that simulates the movement of items through a set of locations in a manner that mimics the operation of a large retailer. Locations have a multinomial transition probability distribution for the likelihood of moving to a given next location or stopping, a multinomial duration probability distribution for the length of stay of items at the location, and a list of correlations with other locations. We also associate a preferred level to locations, to indicate the stage in which they tend to appear. For a given location, transition probabilities are high for locations at the next level and low for locations at any other level, this generates the effect that items tend to move forward in the supply chain but that sometimes can still go backwards or jump to a future location.

As notational convenience, we use the following symbols to denote certain data set parameters. $\mathcal{N}$: Number of paths. ϵ: conditional probability threshold. δ: minimum conditional probability support.

6.4.2 Workflow Compression

One of the main advantages of using a FlowGraph with minimum support δ and minimum conditional probability deviation ϵ is that it captures the essence of the *complete workflow* but requires only a fraction of the space. In this section we compare the size of a FlowGraph with that one of a *complete workflow*. The *complete workflow* is constructed by traversing the clustered path database adding each path to a probabilistic prefix tree, while keeping track of the counts of items reaching each node and the transition counts. We will measure the size $\mathcal{S}$ of a workflow by the total number of probability entries. The size of a FlowGraph is the sum of the number of probabilities for the conditional emission, and conditional transition tables for each node. The size of a *complete workflow* is just the sum of the number of transition probabilities for each node.

Figure 6.4 shows the size of the FlowGraph for various levels of ϵ, for a data set containing 100,000 paths, and using a support δ of 0.1%. In the Figure we distinguish the count of conditional and unconditional transition and emission probabilities. For this experiment the size of the *complete workflow* is of 14,095 probability entries. As we decrease ϵ the number of non-redundant conditional transition and emission probabilities increases. For $\epsilon = 0.2$ we do not record any conditional probabilities and the size of the workflow is just the unconditional emission and transition probabilities, as we decrease ϵ the percentage of the total workflow size accounted by conditional probabilities increases from 0% to around 80%. The size of the FlowGraph is just 1.5% to 7.5% of the size of the *complete workflow*, we observe very significant space savings even for small values of ϵ.

Compression power is even more dramatic when we increase the number of distinct paths, for this example we generated another data set containing every possible distinct path with a count proportional to its likelihood given the location characteristics. The size of the *complete workflow* increased to 1,190,479 nodes, while the size of the

FlowGraph increased only marginally. Compression power for this case increased to 10,000 to 1.

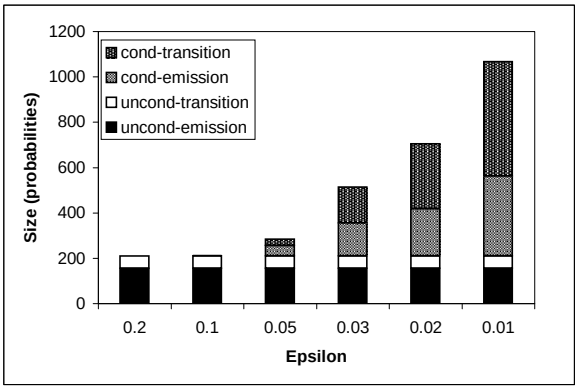


Figure 6.4: Compression vs. Conditional probability threshold (ϵ). $\mathcal{N} = 100,000$, $\delta = 0.001$

Figure 6.5 shows the size of the FlowGraph for the same data set used for Figure 6.4; with a fixed ϵ at 5% and with varying support levels. The support parameter determines the number of frequent closed paths that need to be considered in constructing the conditional probability tables of each node, lower support levels tend to create more conditional probability entries. For this experiment we used support levels of 1% to 0.001%; as we decrease support the percentage of the total workflow size corresponding to conditional probabilities increases from 0% to around 95%. As in the previous case compression is very significant, the size of the FlowGraph is just 1.5% to 4.5% of the size of the *complete workflow*.

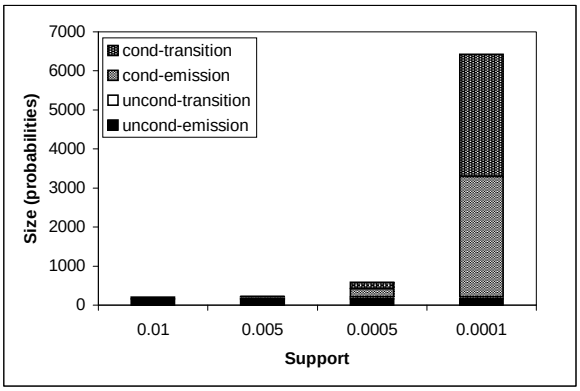


Figure 6.5: Compression vs. Support (δ). $\mathcal{N} = 100,000$, $\epsilon = 0.05$

Figure 6.6 shows the size of the FlowGraph and the *complete workflow* for different sizes of the input clustered path database. We can see that while the size of the FlowGraph remains largely constant the size of the *complete workflow* increases significantly with the number of paths. The reason is that in the FlowGraph we only need to update conditional emission and transition probability entries, while in the *complete workflow* new nodes need to

be added for each new location/duration observed. In this experiment the error induced by compression remained constant for all database sizes at around 9%.

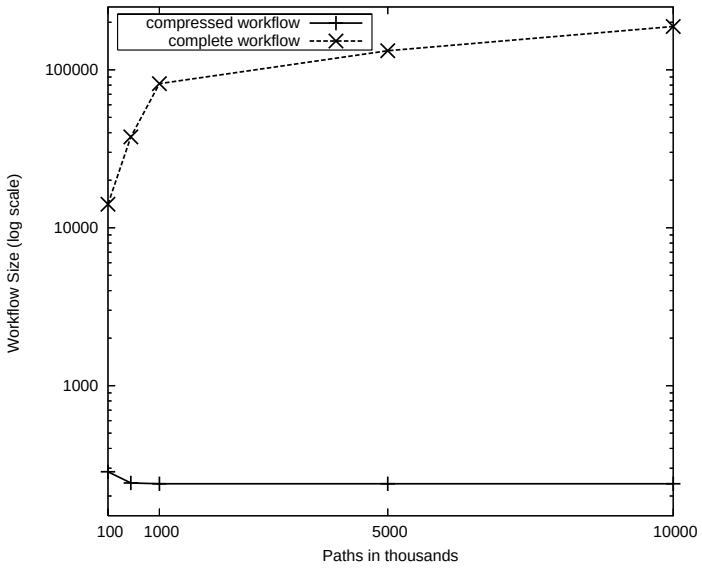


Figure 6.6: Clustered Path DB size vs. Workflow Size. $\epsilon = 0.05$, $\delta = 0.001$

6.4.3 Workflow Compression Error

In this section we look at the error incurred in computing the probability of a path using the FlowGraph as compared to the *complete workflow*. A FlowGraph with probability deviation ϵ and minimum support δ may not be able to assign the exact probability to every possible path, because it only records emission and transition probability deviations if they are supported by more than δ transactions and the deviation is larger than ϵ of the expected probability. The FlowGraph will assign probabilities that may be different from the true probability to uncommon paths that deviate from their expected probabilities (in absolute terms the deviation will still be small as the path will have very low probability to begin with, but in percentual terms it can be large). On the other hand, more common paths, or path segments, will be assigned very precise probabilities, because their deviations will have enough supporting transactions. The experiments on this section correspond to the same data sets used to compute FlowGraph size in the previous section.

For the experiments in this section we will use the *percentual error* measure[4], which computes the percentual deviation in probability from the *complete workflow*, and is defined as,

$$\frac{|P(path|compressed\_wf) - P(path|complete\_wf)|}{P(path|complete\_wf)}$$

[4]We also use KL-Divergence of the computed probabilities and the true probabilities and this measure mimics percentual error.

Figure 6.7 shows the percentual error that a FlowGraph will assign to three sets of paths with varying levels of ϵ. The sets of paths used for testing have a probability of occurring (minimum probability) of 0.0001, 0.0002, and 0.0005 respectively. We can see that as we increase ϵ the error increases, and it increases faster for uncommon paths than for more common ones. In this case the common paths have an error of around 9% with every level of ϵ while the other two sets go from 13% to 17% and 16% to 27% respectively. This means that even with a low ϵ we can compute the probability of more common paths and path segments with low error and storing only a small FlowGraph.

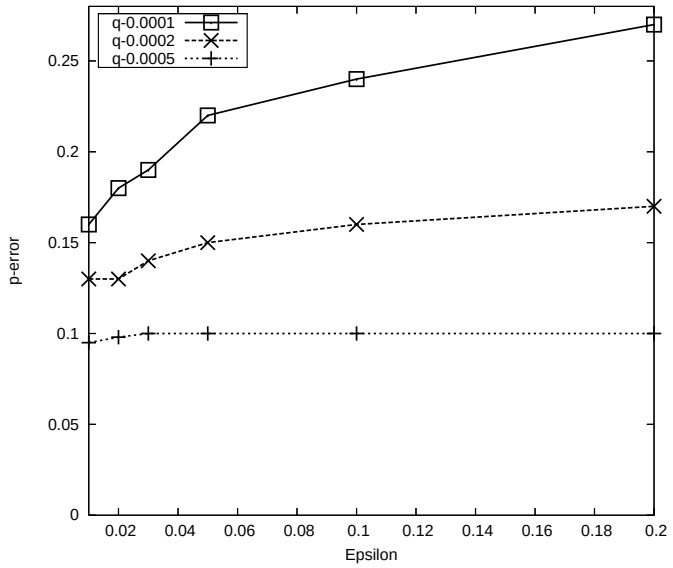


Figure 6.7: Percentual Error vs. Conditional probability threshold (ϵ). $\mathcal{N} = 100,000$, $\delta = 0.001$

Figure 6.8 shows the percentual error that a FlowGraph will assign to the same three sets of paths as before but with varying the support threshold δ, used to compute the closed paths necessary for recording conditional probability entries. We can see that as we increase support, the percentual error grows; faster for the path data sets with low probability. Common paths have an error of around 10% with every level of δ while the other two sets go from 12% to 17% and 14% to 27% respectively. In this case we can also observe that using a small support threshold is enough to compute the probabilities of common paths, and path segments, with low error. As discussed in section 3.2.4 we can use the slope of the error curves to determine good values for ϵ for a particular type of query load.

6.4.4 Construction Time

In this section we analyze the efficiency of the algorithm used to construct the FlowGraph. The main components of the algorithm are: (1) computation of the unconditional emission and transition probabilities, (2) computation of the closed paths for a given support δ, and (3) computation of the conditional emission and transition entries given the frequent closed paths, and a conditional probability threshold ϵ. In our experiments we observed that the factor with

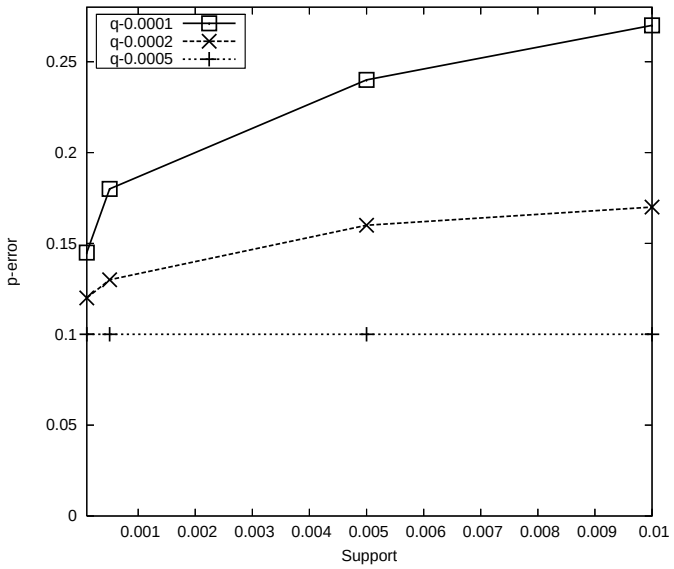


Figure 6.8: Error vs. Support (δ). $\mathcal{N} = 100,000$, $\epsilon = 0.05$

highest influence in computation time was the support threshold, given that it determines how many closed paths will be found, and we need to test each node for conditional dependencies against each of the closed paths.

Figure 6.9 shows the computation of the conditional emission and transition probabilities for a clustered path database with 1,000,000 paths. We can see that as we increase support, the computation time decreases. For this experiment the time to learn the unconditional probability entries of the workflow was unaffected by the support threshold and remained constant at around 4 seconds. For mining closed patterns we used the program closet+ [91], and the run time was around 2 seconds for each support level.

6.5 Summary

In this chapter we have proposed a novel model for the design and construction of a highly compressed RFID workflow that captures the main trends and important deviations in item movements in an RFID application. The FlowGraph records for each node, conditional emission and transition probability tables that have an entry for each non-redundant, and sufficiently supported deviation from the main movement trend. This design takes advantage of the fact that in many RFID applications items tend to move according to a general trend, and have a relatively small number of deviations from it. Our performance study shows that the size of the FlowGraph is only a fraction of that of a *complete workflow*, that it can be used to compute the probabilities of paths and path segments with high precision, and that it can be computed efficiently.

The FlowGraph presented in our study can be a very useful in providing guidance to users in their analysis process

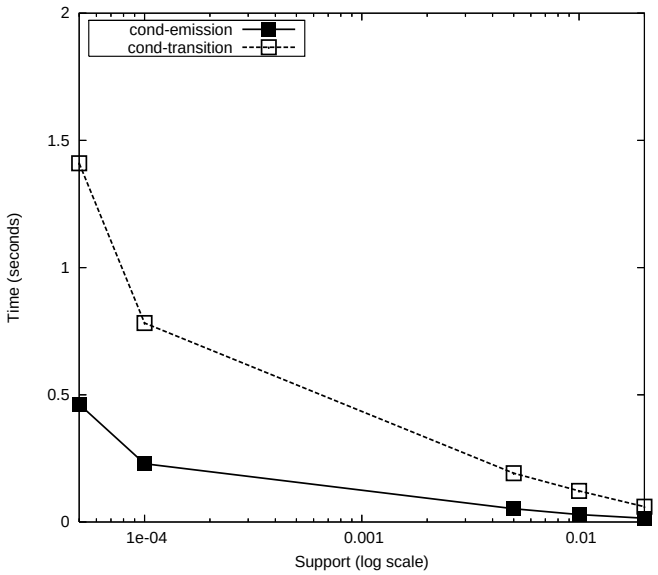


Figure 6.9: Construction Time vs Support (δ). $\mathcal{N} = 1,000,000$, $\epsilon = 0.05$

as it highlights general trends and exceptions which may not be apparent from the raw data. This knowledge can be used to better understand the way objects move through the supply chain, and optimize the logistics process governing the most common flow trends; while exception flow information can be used to uncover events that may for example trigger a higher than expected return rate.

Notice that our proposal for workflow compression is based on the assumption that commodities flow according to a general trend and only few and significant deviations are present in the data. This fits a good number of RFID applications, such as supply chain management. However, there are also other applications where RFID data may not have such characteristics. We believe that further research is needed to construct efficient workflow models for such applications.

Chapter 7

FlowCube Mining

In the previous chapter we developed the concept of FlowGraphs and proposed an efficient computation algorithm. In this chapter we extend the FlowGraph concept to a multi-dimensional FlowCube model, and bring the power of OLAP to commodity flow analysis. The FlowCube can be used for example, to look at the evolution of the FlowGraphs for a certain product category over a period of time, to detect how a change in suppliers may have affected the probability of returns for a particular item. We could also use multidimensional analysis to compare the speed at which products from two different manufacturers move through the system, and use that information to improve inventory management policies. Furthermore, it may be interesting to look at paths traversed by the items from different perspectives. A transportation manager may want to look at FlowGraphs that provide great detail on truck, distribution centers, and sorting facilities while ignoring most other locations. A store manager on the other hand may be more interested in looking at movements from backrooms, to shelfs, checkout counters, and return counters and largely ignore other locations.

7.1 FlowCube

In this section we will introduce the concept of a FlowCube, which is a data cube computed on an RFID path database, where each cell summarizes commodity flows at at a given abstraction level of the path independent dimensions, and path stages. The measure recorded in each cell of the FlowCube is a FlowGraph computed on the paths belonging to the cell.

In the next sections we will explore in detail the different components of a FlowCube. We will first introduce the concepts of item abstraction lattice and path abstraction lattice, which are important to give a more precise definition of the cuboid structure of a FlowCube. We will then study the computational challenges of using FlowGraphs as measures. Finally we introduce the concepts of non-redundant FlowCubes, and iceberg flow-cubes as a way to reduce the size of the model.

Table 7.1 presents a path database with 2 path independent dimensions: product and brand. The nomenclature used for stage locations is d for distribution center, t for truck, w for warehouse, s for store shelf, c for store checkout,

and f for factory.

id	product	brand	path
1	tennis	nike	$(f, 10)(d, 2)(t, 1)(s, 5)(c, 0)$
2	tennis	nike	$(f, 5)(d, 2)(t, 1)(s, 10)(c, 0)$
3	sandals	nike	$(f, 10)(d, 1)(t, 2)(s, 5)(c, 0)$
4	shirt	nike	$(f, 10)(t, 1)(s, 5)(c, 0)$
5	jacket	nike	$(f, 10)(t, 2)(s, 5)(c, 1)$
6	jacket	nike	$(f, 10)(t, 1)(w, 5)$
7	tennis	adidas	$(f, 5)(d, 2)(t, 2)(s, 20)$
8	tennis	adidas	$(f, 5)(d, 2)(t, 3)(s, 10)(d, 5)$

Table 7.1: Path database

7.1.1 Abstraction Lattice

Each dimension in the flow cube can have an associated concept hierarchy. A concept hierarchy is a tree where nodes correspond to concepts, and edges correspond to *is-a* relationships between concepts. The most concrete concepts reside at the leafs of the tree, while the most general concept, denoted '*', resides at the apex of the tree and represents *any* concept. The level of abstraction of a concept in the hierarchy is the level at which the concept is located in the tree.

Item Lattice. The abstraction level of the items in the path database can be represented by the tuple $(l_1, ..., l_m)$, where l_i is the abstraction level of the path independent dimension d_i. For our running example we can say that the items in the path database presented reside at the lowest abstraction level. The set of all item abstraction levels forms a lattice. A node n_1 is higher in the lattice than a node n_2, denoted $n_1 \preceq n_2$ if the levels all dimensions in n_1 are smaller or equal to the ones in n_2.

Table 7.2 shows the path independent dimension from Table 7.1 with the product dimension aggregated one level higher in its concept hierarchy. The "path ids" column lists the paths in the cell, each number corresponds to the path id in Table 7.1. We can compute a FlowGraph on each cell in Table 7.2. Figure 7.1 presents the FlowGraph for the cell (outerwear, nike).

product	brand	path ids
shoes	nike	1,2,3
shoes	adidas	7,8
outerwear	nike	4,5,6

Table 7.2: Aggregated path database

Path Lattice. In the same way that items can be associated with an abstraction level, path stages will also reside at some level of the location and duration concept hierarchies. Figure 7.2 presents an example concept hierarchy for the location dimension of the path stages. The shadowed nodes are the concepts that are important for analysis; in this

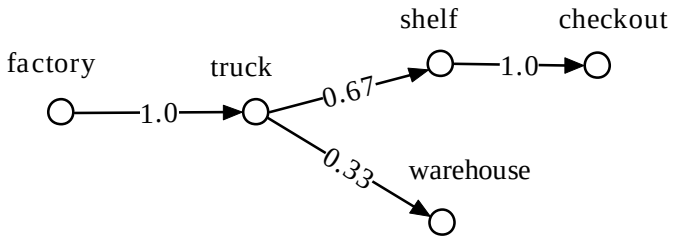


Figure 7.1: Flowgraph for cell (outerwear, nike, 99)

case the data analyst may be a transportation manager that is interesting in seeing the transportation locations at a full level of detail, while aggregating store and factory locations to a higher level. More formally, the path abstraction level is defined by the tuple $(\langle v_1, v_2, ..., v_k \rangle, t_l)$ where each v_i is a node in the location concept hierarchy, and t_l the level in the time concept hierarchy. Analogously to the item abstraction lattice definition, we can define a path abstraction lattice.

In our running example, assuming that time is at the hour level, the path abstraction level corresponding to Figure 7.2 is ($\langle$dist. center, truck, warehouse, factory, store$\rangle$,hour).

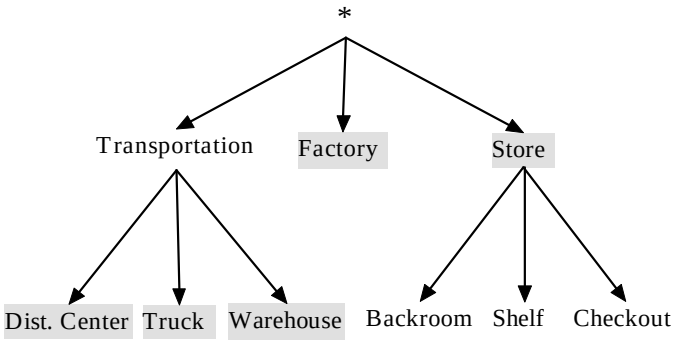


Figure 7.2: Location concept hierarchy

We aggregate a path to abstraction level $(\langle v_1, v_2, ..., v_k \rangle, t_l)$ in two steps. First, we aggregate the location in each stage to its corresponding node v_i, and we aggregate its duration to the level t_l. Second, we merge consecutive locations that have been aggregated to the same concept level. The merging of consecutive locations requires us to define a new duration for the merged stage. The computation of the merged duration would depend on the application, it could be as simple as just adding the individual durations, or it could involve some form of numerosity reduction based on clustering or other well known methods.

Aggregation along the path abstraction lattice is new to FlowCubes and is quite different to the type of aggregation performed in a regular data cube. In a data cube, an aggregated cell contains a measure on the subset of tuples from the fact table that share the same values on every aggregated dimension. When we do path aggregation, the dimensions

from the fact table remain unchanged, but it is the measure of the cell itself which changes. This distinct property requires us to develop new methods to construct a FlowCube that has aggregation for both item and path dimensions.

Definition 7.1.1 (Flowcube) *A FlowCube is a collection of cuboids. A cuboid is a grouping of entries in the fact table into cells, such that each cell shares the same values on the item dimensions aggregated to an item abstraction level I_l; and the paths in the cell have been aggregated to a path abstraction level P_l. The measure of a cell is a FlowGraph computed on the paths in the cell. A cuboid can be characterized by the pair $\langle I_l, P_l \rangle$.*

7.1.2 Measure Computation

We can divide a FlowGraph into two components, the first is the duration and transition probability distributions, the second is the set of exceptions. In this section we will show that while the first component is an algebraic measure, and thus can be computed efficiently, the second component is a holistic measure and requires special treatment.

Assume that we have a dataset S that has been partitioned into k subsets $s_1, ..., s_k$ such that $S = \bigcup_i s_i$ and $s_i \cap s_j = \phi$ for all $i \neq j$. We say that a measure, or function, is algebraic if the measure for S can be computed by collecting M (positive bounded integer) values from each subset s_i. For example, average is distributive as we can collect count() and sum() ($M = 2$) from each subset to compute the global average. A holistic measure on the other hand is one where there is no constant bound on the number of values that need to be collected from each subset in order to compute the function for S. Median is an example of a holistic function, as each subset will need to provide its entire set of values in order to compute the global median.

Lemma 7.1.2 *The duration and transition distributions of a FlowGraph are algebraic measures.*

Proof Sketch. Each node n in the FlowGraph contains a duration probability distribution d and a transition probability distribution t. For each node in the FlowGraph, $d(t_i) = count(t_i) / \sum_{i=1}^{k} count(t_i)$, where $d(t_i)$ is the probability of duration t_i, $count(t_i)$ is the number of items that stayed at the node for duration t_i and k is the number of distinct durations at the node. We can compute d for each node in a FlowGraph whose dataset has been partitioned into subsets, by collecting the following k values $count(t_1), ..., count(t_k)$. Similarly we can argue that the transition distribution t can be computed by collecting a fixed number of transition counts from each subset. Given that the number of nodes and distinct durations per node in a FlowGraph is fixed (after numerosity reduction), we can collect a bounded number of counts from each subset to compute the duration and transition distributions for the FlowGraph. ∎

The implication of lemma 7.1.2 is that we can compute a FlowCube efficiently by constructing high level FlowGraphs from already materialized low level ones without having to go to the path database.

Lemma 7.1.3 *The set of exceptions in a FlowGraph is a holistic measure.*

Proof Sketch. The FlowGraph exceptions are computed on the frequent itemsets in the collection of paths aggregated in the cell, thus proving that a function that returns the frequent itemsets for a cell is not algebraic is sufficient. Assume that the set S is the union of $s_1, ..., s_n$, and that the sets $f_1, ..., f_n$ are the frequent itemsets for each subset s_i. We need to compute F the frequent itemsets for S, assume that f_i^j is frequent pattern j in set i, in order to check if f_i^j is frequent in S we need to collect its count on every subset s_k, and thus we need every subset to provide counts for every frequent pattern on any other subset, this number is clearly unbounded as it depends on the characteristics of each data set. ∎

The implication of lemma 7.1.3 is that we can not compute high level FlowGraphs from low level ones by just passing a fixed amount of information between the levels. But we can still mine the frequent patterns required to determine exceptions in each cell in a very efficient way by entirely avoiding the level by level computation approach and instead using a novel shared computation method that simultaneously finds frequent patterns for cuboids at every level of abstraction. In section 5 we will develop the mining method in detail.

7.1.3 FlowGraph Redundancy

The FlowGraph registered for a given cell in a FlowCube may not provide new information on the characteristics of the data in the cell, if the cells at a higher abstraction level on the item lattice, and the same abstraction level on the path lattice, can be used to derive the FlowGraph in the cell. For example, if we have a FlowGraph G_1 for milk, and a FlowGraph G_2 from milk 2% (milk is an ancestor of milk 2% in the item abstraction lattice), and $G_1 = G_2$ we see that G_2 is redundant, as it can be inferred from G_1.

Before we give a more formal definition of redundancy we need a way to determine the similarity of two FlowGraphs. A similarity metric between two FlowGraphs is a function $\varphi : G_1 \times G_2 \rightarrow \mathbb{R}$. Informally the value of $\varphi(G_1, G_2)$ is large if the G_1 and G_2 are similar and small otherwise. There are many options for φ and the one to use should use depends on the particular RFID application semantics. One possible funtion is to use the KL-Divergence of the probability distributions induced by two FlowGraphs. But other similarity metrics, based for example on probabilistic deterministic finite automaton (PDFA) distance could be used. Note that we do not require φ to be a real metric in the mathematical sense, in that the triangle inequality does not necessarily need to hold.

Definition 7.1.4 (Redundant FlowGraph) *Let G be a FlowGraph for cell c, let $p_1, ..., p_n$ be all the cells in the item lattice that are a parent of c and that reside in a cuboid at the same level in the path lattice as c's cuboid. Let $G_1, ..., G_n$ be the FlowGraphs for cells $p_1, ..., p_n$, let φ be a FlowGraph similarity metric. We say that G is redundant if $\varphi(G, G_i) > \tau$ for all i, where τ is the similarity threshold.*

A FlowCube that contains only cells with non-redundant FlowGraphs is called a non-redundant FlowCube. A

non-redundant FlowCube can provide significant space savings when compared to a complete FlowCube, but more interestingly, it provides important insight into the relationship of flow patterns from high to low levels of abstraction, and can facilitate the discovery of exceptions in multi-dimensional space. For example, using a non-redundant FlowCube we can quickly determine that milk from every manufacturer has very similar flow patterns, except for the milk from farm A which has significant differences. Using this information the data analyst can drill down and slice on farm A to determine what factors make its FlowGraph different.

7.1.4 Iceberg Flowcube

A FlowGraph is a statistical model that describes the flow behavior of objects given a collection of paths. If the data set on which the FlowGraph is computed is very small, the FlowGraph may not be useful in conducting data analysis. Each probability in the model will be supported by such a small number of observations and it may not be an accurate estimate of the true probability. In order to minimize this problem, we will materialize only cells in the FlowCube that contain at least δ paths (minimum support). For example, if we set the minimum support to 2, the cell $(shirt, *)$ from Table 7.1 will not be materialized as it contains only a single path.

Definition 7.1.5 (Iceberg FlowCube) *A FlowCube that contains only cells with a path count larger than δ is called an Iceberg FlowCube.*

Iceberg FlowCubes can be computed efficiently by using apriori pruning of infrequent cells. We can materialize the cube from low abstraction levels to high abstraction ones. If at some point a low level cell is not frequent, we do not need to check the frequency of any specialization of the cell. The algorithm we develop in the next section will make extensive use of this property to speed up the computation of the FlowCube.

7.2 Algorithms

In this section we will develop a method to compute a non-redundant iceberg FlowCube given an input path database. The problem of FlowCube construction can be divided into two parts. The first is to compute the FlowGraph for each frequent cell in the cube, and the second is to prune uninteresting cells given higher abstraction level cells. The second problem can be solved once the FlowCube has been materialized, by traversing the cuboid lattice from low to high abstraction levels, while pruning cells that are found to be redundant given the parents. In the rest of this chapter we will focus on solving the first problem.

The key computational challenge in materializing a FlowCube is to find the set of frequent path segments, aggregated at every interesting abstraction level, for every cell that appears frequently in the path database. Once we

have the counts for every frequent pattern in a cell determining exceptions can be done efficiently by just checking if counts of these patterns change the duration or transition probability distributions for each node. The problem of mining frequent patterns in the FlowCube is very expensive as we need to mine frequent paths at every cell, and the number of cells is exponential in the number of item and path dimensions. Flowcube materialization combines two of the most expensive methods in data mining, cube computation, and frequent pattern mining. The method that we develop in this section solves these two problems with a modified version of the Apriori algorithm [3], by collecting frequent pattern counts at every interesting level of the item and path abstraction lattices simultaneously, while exploiting cross-pruning opportunities between these two lattices to reduce the search space as early as possible. To further improve performance our algorithm will use partial materialization to restrict the set of cuboids to compute, to those most useful to each particular application. The algorithm is based on the following key ideas:

Construction of a transaction database. In order to run a frequent pattern algorithm on both the item and path dimensions at every abstraction level, we need to transform the original path database into a transaction database. Values in the path database need to be transformed into items that encode their concept hierarchy information and thus facilitate efficient multi-level mining. For example, the value "jacket" for the product dimension in Table 7.1, can be encoded as "112", the first digit indicates that it is a value of the first path independent dimension, the second digit indicates that is of type outerwear, and the third digit tells us that it is a jacket (for brevity we omit the encoding for product category as all the products in our example belong to the same category: clothing). Path stages require a slightly different encoding, in addition to recording the concept hierarchy for the location and time dimensions for the stage, each stage should also record the path prefix leading to the stage so that we can do multi-level path aggregation. For example the stage (t,1) in the first path of the path database in Table 7.1 can be encoded as (fdt,10), to mean that it is the third stage in the path: factory → dist. center → truck, and that it has a duration of 10 time units.

Table 7.3 presents the transformed database resulting from the path database from Table 7.1.

TID	Items
1	{121,211,(f,10),(fd,2),(fdt,1),(fdts,5),(fdtsc,0)}
2	{121,211,(f,5),(fd,2),(fdt,1),(fdts,10),(fdtsc,0)}
3	{122,211,(f,10),(fd,1),(fdt,2),(fdts,5),(fdtsc,0)}
4	{111,211,(f,10),(ft,1),(fts,5),(ftsc,0)}
5	{112,211,(f,10),(ft,2),(fts,5),(ftsc,1)}
6	{112,211,(f,10),(ft,1),(ftw,5)}
7	{121,221,(f,5),(fd,2),(fdt,2),(fdts,20)}
8	{121,221,(f,5),(fd,2),(fdt,3),(fdts,10),(fdtsd,5)}

Table 7.3: Transformed transaction database

Shared counting of frequent patterns. In order to minimize the number of scans of the transformed transaction database we share the counting of frequent patterns at every abstraction level in a single scan. Every item that we encounter in a transaction contributes to the support of all of its ancestors on either the item or path lattices. For

example, the item 112 (jacket) contributes to its own support and the support of its ancestors, namely, 11* (outerwear) and 1** (we will later show that this ancestor is not really needed). Similarly an item representing a path stage contributes to the support of all of its ancestors along the path lattice. For example, the path stage (fdts,10) will support its own item and items such as (fdts,*), (fTs,10) and (fTs,*), where f stands for factory, d for distribution center, t for truck, s for shelf, and T for transportation (T is the parent of d, and t, in the location concept hierarchy).

Shared counting processes patterns from short to long. In the first scan of the database we can collect all the patterns of length 1, at every abstraction level in the item and path lattices. In the second scan we check the frequency of candidate patterns of length 2 (formed by joining frequent patterns of length 1). We continue this process until no more frequent patterns are found. Table 7.4 presents a portion of the frequent patterns of length 1 and length 2 for the transformed path database of Table 7.3.

Length 1 frequent

Itemset	Support
{121}	5
{12*}	5
{(f,10)}	5
{(f,*)}	8
{(fd,2)}	4
...	...

Length 2 frequent

Itemset	Support
{12*,211}	3
{12*,21*}	3
{211,(f,10)}	4
{(f,5)(fd,2)}	3
{(f,*),(fd,*)}	3
...	...

Table 7.4: Frequent itemsets

Pruning of infrequent candidates. When we generate candidates of length $k+1$ based on frequent patterns of length k we can apply several optimization techniques to prune large portions of the candidate space:

- **Precounting of frequent itemsets at high levels of abstraction along the item and path lattices**. We can take advantage of the fact that infrequent itemsets at high abstraction levels of the item and path lattice can not be frequent at low abstraction levels. We can implement this strategy by, for example, counting frequent patterns of length 2 at a high abstraction level while we scan the database to find the frequent patterns of length 1. A more general precounting strategy could be to count high abstraction level patterns of length $k+1$ when counting the support of length k patterns.

- **Pruning of candidates containing two unrelated stages**. Given our stage encoding, we can quickly determine if two stages can really appear in a the same path, and prune all those candidates that contain stages that can not appear together. For example we know that the stages $(fd, 2)$ and $(fts, 5)$ can never appear in the same path, and thus should not be generated as a candidate.

- **Pruning of path independent dimensions aggregated to the highest abstraction level**. We do not need to collect counts for an item such as 1**, this item really mean any value for dimension 1; and its count will

always be the same as the size of the transaction table. These type of items can be removed from the transaction database.

- **Pruning of items and their ancestors in the same transaction**. This optimization was introduced in [85] and is useful for our problem. We do not need to count an item and any of its ancestors in the same candidate as we know that the ancestor will always appear with the item. This optimization can be applied to both path independent dimension values, and path stages. For example we should not consider the candidate itemset $\{121, 12*\}$ as its count will be the same of the itemset $\{121\}$.

Partial Materialization. Even after applying all the optimizations outlined above, and removing infrequent and redundant cells the size of of FlowCube can still be very large in the cases when we have a high dimensional path database. Under such conditions we can use the techniques of partial materialization developed for traditional data cubes [52, 50, 84].

One strategy that seems especially well suited to our problem is that of partial materialization described in [50], which suggests the computation of a layer of cuboids at a minimum abstraction level that is interesting to users, a layer at an observation level where most analysis will take place, and the materialization of a few cuboids along popular paths in between these two layers.

7.2.1 Shared Algorithm

Based on the optimization techniques introduced in the previous section, we propose algorithm Shared which is a modified version of the Apriori algorithm [3] used to mine frequent itemsets. Shared simultaneously computes the frequent cells, and the frequent path segments aggregated at every interesting abstraction level along the item and path lattices. The output of the algorithm can be used to compute the FlowGraph for every cell that passes the minimum support threshold in the FlowCube.

7.2.2 Cubing Based Algorithm

A natural competitor to the Shared algorithm is an iceberg cubing algorithm that computes only cells that pass the iceberg condition on the item dimensions, and that for each such cell calls a frequent pattern mining algorithm to find frequent path segments in the cell. The precise cubing algorithm used in this problem is not critical, as long as the cube computation order is from high abstraction level to low level, because such order enables early pruning of infrequent portions of the cube. Examples of algorithms that fall into this category are BUC [7] and Star Cubing [92].

Algorithm 6 takes advantage of pruning opportunities based on the path independent dimensions, *i.e.*, if it detects that a certain value for a given dimension is infrequent, it will not check that value combined with another dimension

Algorithm 5 Shared

Input: A path database D, a minimum support δ
Output: Frequent cells and frequent path segments in every cell
Method:

1: In one scan of the path database compute the transformed path database into D', collect frequent items of length 1 into L_1, and pre-count patterns of length > 1 at high abstraction levels into P_1.
2: **for** $k = 2$,$L_{k-1} \neq \phi$,$k++$ **do**
3: generate C_k by joining frequent patterns in L_{k-1}
4: Remove from C_k candidates that are infrequent given the pre-counted set P_{k-1}, remove candidates that include stages that can not be linked, and remove candidates that contain an item and its ancestor.
5: **for** every transaction t in D' **do**
6: increment the count of candidates in C_k supported by t, and collect the counts of high abstraction level patterns of length $> k$ into P_k
7: **end for**
8: L_k = frequent items in C_k
9: **end for**
10: Return $\bigcup_k L_k$.

Algorithm 6 Cubing

Input: A path database D, a minimum support δ
Output: Frequent cells and frequent path segments in every cell
Method:

1: Divide D into two components D_i, which contains the path independent dimensions, and D_p which contains the paths.
2: Transform D_p into a transaction database by encoding path stages into items, and assign to each transaction a unique identifier.
3: Compute the iceberg cube C on D_i, use as measure the list of transaction identifiers aggregated in the cell.
4: **for** each cell c_i in C **do**
5: c_i^p = read the transactions aggregated in the cell.
6: c_i^f = find frequent patterns in c_i^p by using a frequent pattern mining algorithm
7: **end for**
8: return $\bigcup_i c_i^f$.

because it is necessarily infrequent. What the algorithm misses is the ability to do pruning based on the path abstraction lattice. It will not, for example, detect that a certain path stage is infrequent in the highest abstraction level cuboid and thus will be infrequent in every other cuboid, the algorithm will repeatedly generate that path stage as a candidate and check its support just to find that it is infrequent every single time. Another disadvantage of the cubing based algorithm is that it has to keep long lists of transaction identifiers as measures for the cells, when the lists are long, the input output costs of reading them can be significant. In our experiments even for moderately sized data sets these lists where much larger than the path database itself. With the algorithm shared, we only record frequent patterns, and thus our input output costs are generally smaller.

7.3 Experimental Evaluation

In this section, we perform a thorough analysis our proposed algorithm (shared) and compare its performance against a baseline algorithm (basic), and against the cubing based algorithm (cubing) presented in the previous section. All experiments were implemented using C++ and were conducted on an Intel Pentium IV 2.4GHz System with 1GB of RAM. The system ran Debian Sarge with the 2.6.13.4 kernel and gcc 4.0.2.

7.3.1 Data Synthesis

The path databases used for our experiments were generated using a synthetic path generator that simulates the movement of items in a retail operation. We first generate the set of all valid sequences of locations that an item can take through the system. Each location in a sequence has an associated concept hierarchy with 2 levels of abstraction. The number of distinct values and skew per level are varied to change the distribution of frequent path segments. The generation of each entry in the path database is done in two steps. We first generate values for the path independent dimensions. Each dimension has a 3 level concept hierarchy. We vary the number of distinct values and the skew for each level to change the distribution of frequent cells. After we have selected the values for the path independent dimensions, we randomly select a valid location sequence from the list of possible ones, and generate a path by assigning a random duration to each location. The values for the levels in the concept hierarchies for path independent dimensions, stage locations, and stage durations, are all drawn from a Zipf distribution [95] with varying α to simulate different degrees of data skew.

For the experiments we compute frequent patterns for every cell at every abstraction level of the path independent dimensions, and for path stages we aggregate locations to the level present in the path database and one level higher, and we aggregate durations to the level present in the path database and to the any (*) level, for a total of 4 path abstraction levels.

In most of the experiments we compare three methods: Shared, Cubing, and Basic. Shared is the algorithm that we propose in section 5.1 and that does simultaneous mining of frequent cells and frequent path segments at all abstraction levels. For shared we implemented pre-counting of frequent patterns of length 2 at abstraction level 2, and pre-counting of path stages with duration aggregated to the '*' level. Cubing is an implementation of the algorithm described in section 5.2, we use a modified version of BUC [7] to compute the iceberg cube on the path independent dimensions and then called Apriori [3] to mine frequent path segments in each cell. Basic is the same algorithm as Shared except that we do not perform any candidate pruning based on the optimizations outlined in the previous section. In the figures we will use the following notation to represent different data set parameters $\mathcal{N}$ for the number of records, δ for minimum support, and d for the number of path independent dimensions.

7.3.2 Path Database Size

In this experiment we look at the runtime performance of the three algorithms when varying the size of path database, from 100,000 paths to 1,000,000 paths (disk size of 6 megabytes to 65 megabytes respectively). In Figure 7.3 we can see that the performance of shared and cubing is quite close for smaller data sets but as we increase the number of paths the runtime of shared increases with a smaller slope than that of cubing. This may be due to the fact that as we increase the number of paths the data set becomes denser BUC slows down. Another influencing factor in the difference in slopes is that as the data sets become denser cubing needs to invoke the frequent pattern mining algorithm for many more cells, each with a larger number of paths. We were able to run the basic algorithm for 100,000 and 200,000 paths, for other values the number of candidates was so large that they could not fit into memory.

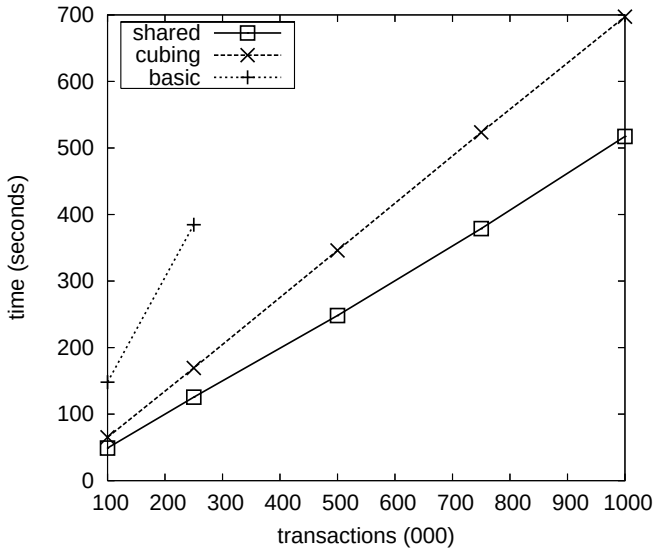


Figure 7.3: Database size ($\delta = 0.01$, $d = 5$)

7.3.3 Minimum Support

In this experiment we constructed a path database with 100,000 paths and 5 path independent dimensions. We varied the minimum support from 0.3% to 2.0%. In Figure 7.4 we can see that shared outperforms cubing and basic. As we increase minimum support the performance of all the algorithms improves as expected. Basic improves faster that the other two, this is due to the fact that fewer candidates are generated at higher support levels, and thus optimizations based on candidate pruning become less critical. For every support level we can see that shared outperforms cubing, but what is more important we see that shared improves its performance faster than cubing. The reason is that as we increase support shared will quickly prune large portions of the path space, while cubing will repeatedly check this portions for every cell it finds to be frequent.

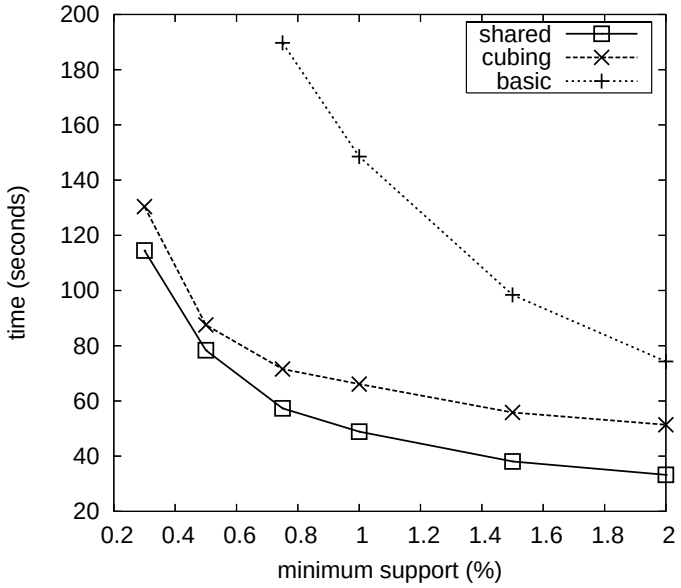


Figure 7.4: Minimum support ($\mathcal{N} = 100,000$, $d = 5$)

7.3.4 Number of Dimensions

In this experiment we kept the number of paths constant at 100,000 and the support at 1%, and varied the number of dimensions from 2 to 10. The datasets used for this experiment were quite sparse to prevent the number of frequent cells to explode at higher dimension cuboids. The sparse nature of the datasets makes all the algorithms achieve a similar performance level. We can see in Figure 7.5 that both shared and cubing are able to prune large portions of the cube space very soon, and thus performance was comparable. Similarly basic was quite efficient as the number of candidates was small and optimizations based on candidate pruning did not make a big difference given that the number of candidates was small to begin with.

7.3.5 Density of the Path Independent Dimensions

For this experiment we created three datasets with varying numbers of distinct values in 5 path independent dimensions. Dataset a had 2, 2, and 5 distinct values per level in every dimension; dataset b has 4, 4, and 6; dataset c has 5, 5, and 10. In Figure 7.6 we can see that as we increase the number of distinct items, data sparsity increases, and fewer frequent cells and path segments are found, which significantly improves the performance of all three algorithms. Due to the very large number of candidates we could not run the basic algorithm for dataset a.

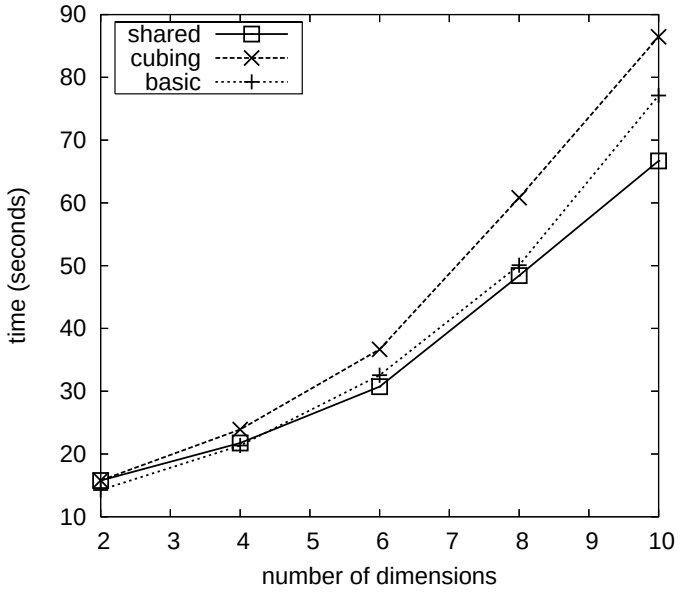


Figure 7.5: Number of dimensions. ($\mathcal{N} = 100,000$, $\delta = 0.01$)

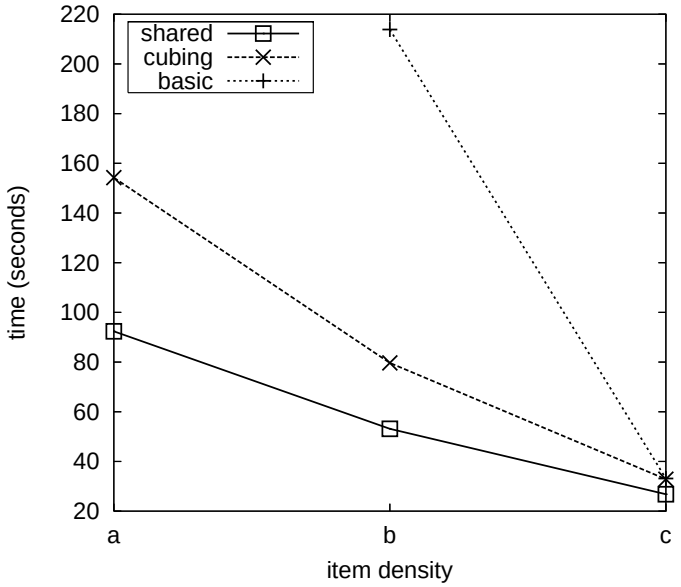


Figure 7.6: Item density ($\mathcal{N} = 100,000$, $\delta = 0.01$, $d = 5$)

7.3.6 Density of the Paths

In this experiment we kept the density of the path independent dimensions constant and varied the density of the path stages by varying the number of distinct location sequences 10 to 150. We can see in Figure 7.7 that for small numbers of distinct path sequences, we have many frequent path fragments and thus mining is more expensive. But what is more important is that as the path database becomes denser shared gains a very significant advantage over cubing. The

reason is that in a few scans of the database shared is able to detect every frequent path segment at every abstraction level, while cubing needs to do find frequent path segments independently for each frequent cell, and given the high density of paths, mining of frequent path segments is an expensive operation. We could not run the basic algorithm on this experiment as the number of candidates exploded with dense paths.

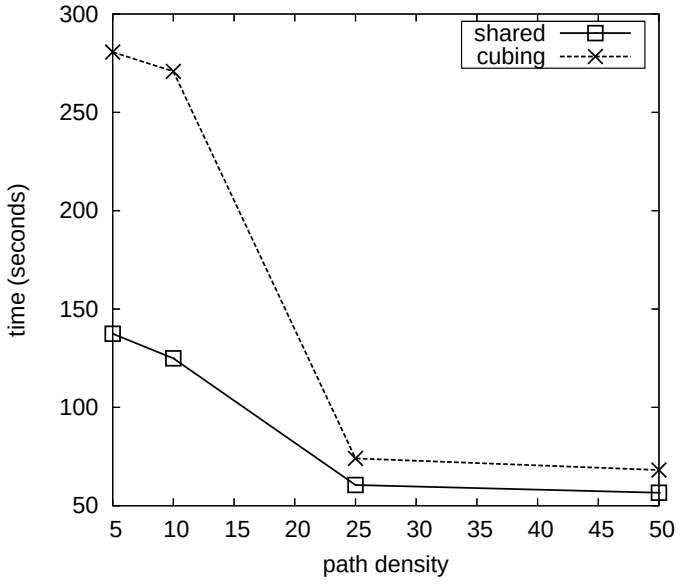


Figure 7.7: Path density ($\mathcal{N} = 100,000$, $\delta = 0.01$, $d = 5$)

7.3.7 Pruning Power

This is experiment we show the effectiveness of the optimizations described in section 5 to prune unpromising candidates from consideration in the mining process. We compare the number of candidates that the basic and shared algorithms need to count for each pattern length. We can see in Figure 7.8 that shared is able to prune a very significant number of candidates from consideration. Basic on the other hand has to collect counts for a very large number of patterns that end up being infrequent, this increases the memory usage and slows down the algorithm. We can also see in the figure that shared considers patterns only up to length 8, while basic considers patterns all the way to length 12. This is because basic is considering long transactions that include items and their ancestors.

In this section we have verified that the ideas of shared computation, simultaneous mining of frequent patterns, and pruning techniques based on both item and path abstraction lattices are effective in practice and provide significant cost savings versus alternative algorithms.

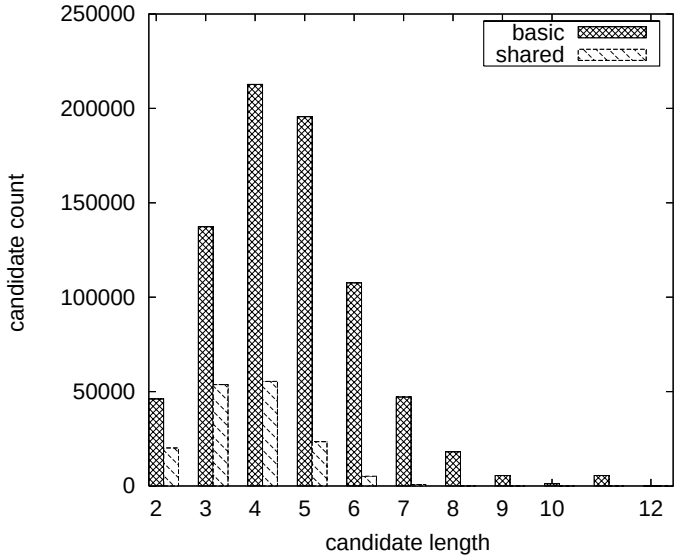


Figure 7.8: Pruning power ($\mathcal{N} = 100,000$, $\delta = 0.01$, $d = 5$)

7.4 Summary

In this chapter we introduced the problem of constructing a FlowCube for a large collection of paths. The FlowCube is data cube model useful in analyzing item flows in an RFID application by summarizing item paths along the dimensions that describe the items, and the dimensions that describe the path stages.

The FlowCube is a very useful tool in providing guidance to users in their analysis process. It facilitates the discovery of trends in the movement of items at different abstraction levels. It also provides views of the data that are tailored to the needs of each user. The FlowCube is particularly well suited for the discovery of exceptions in flow trends, as it only stores non-redundant FlowGraphs that by definition deviate from their ancestor FlowGraphs.

We developed an efficient method to compute the FlowCube based on the ideas of shared computation of frequent flow patterns at every level of abstraction of the item and path lattices. Pruning of the search space by taking taking advantage of the relation between the path and the item view on RFID data. Compression of the cube by the removal of infrequent cells, and redundant FlowGraphs. And partial materialization of high dimensional FlowCubes based on popular cuboids.

Through an empirical study we verify the feasibility of our model and materialization methods. We compared the performance of our proposed algorithm with the performance of two competing algorithms and showed that our solution achieves better performance than those methods under a variety data sizes, data distributions, and minimum support considerations.

Chapter 8

Mining Route Recommendations

Modern highway networks provide several mechanisms for automatic vehicle identification. The most common are the use of toll collection transponders to detect vehicles at multiple points in the network, and the use of cameras to automatically identify license plates. Such information provides valuable patterns useful to online navigation systems and route planning applications. Most existing route planning applications use a fastest path algorithm based on static or dynamic models of road speeds, but such models in general disregard observed driver behavior, and other important factors such as weather, car-pool availability, or vehicle type. Existing solutions may, for example, provide a route that is the fastest one, but that goes through a high crime area, and is thus avoided by experienced drivers. We propose a traffic-mining-based path-finding method [43] that mines speed and driving models from historic traffic data, and uses them to compute fast routes that are well supported by historic driving behavior under the set of relevant driving and traffic conditions.

8.1 Problem Definition

In this section we first define a set of key concepts: *road network, driving patterns, speed patterns*, and *forecast functions*, and then present the problem statement.

Definition 8.1.1 *A road network is a directed graph $G(V, E)$, where V is a set of vertices representing road intersections and terminal points, and E is a set of edges representing road segments each connecting two vertices.*

Figure 8.1 presents the road network for the town of San Joaquin in California. Larger roads are shown in bold. The graph contains 24,123 edges and 18,496 nodes, and was obtained from TIGER line files provided by the US census [1].

Definition 8.1.2 *A speed pattern is a tuple of the form $\langle edege\_id, t\_start, t\_end, (d_1, d_2, \ldots, d_k) : m\rangle$, where $edge\_id$ is an edge, (t_start, t_end) is a time interval, each d_i is a value for speed factor D_i, and m is an aggregate function computed on edge speed.*

[1] http://www.census.gov/geo/www/tiger/tgrcd108/tgr108cd.html

Time	Weather	Vehicle	Speed
8am-10am	Good	Car	45 mph
8am-10am	Bad	Car	35 mph
8am-10am	Good	Truck	40 mph
8am-10am	Bad	Truck	25 mph
10am-5pm	Good	Car	65 mph
...	...	...	...

Table 8.1: Speed pattern for a single edge

Speed patterns describe road speeds under a variety of conditions. Table 8.1 presents an example of speed patterns for a particular edge (road segment) in a road network. In the example we list edge speeds for three factors: *time-of-day,* D_1 = *weather, and* D_2 = *vehicle-type*. In reality, it is possible to consider many more factors, such as road construction and accidents at nearby edges. The speed table can be constructed by integrating information from multiple sources, *e.g.*, we can obtain information on road speed, based on time, location, weather data, and road construction information. Currently, it is possible to obtain speed conditions for roads at certain important metropolitan areas like San Francisco and Chicago, and there is a trend for an increased availability of such information, evidenced by the recent incorporation of speed conditions into major route planning software from Google and Microsoft. Information on speed-affecting factors such as weather and road construction are readily available for most areas of the United States; and other factors such as accident data are expected to become available in the near future.

Definition 8.1.3 *A driving pattern is a sequence* s *of edges* $e_{(1)}, e_{(2)}, \ldots, e_{(l)}$ *that appears more than* min_sup *times in the path database, and that is a valid path in the road network graph* $G(V, E)$*. We define* $support(s)$ *as the number of paths that contain the sequence* s*. We define the length of the sequence* $length(s)$ *as the number of edges that it contains.*

Driving patterns are edge sequences that are frequently traversed by drivers. A path database is a set of trajectories, one per driving session. Currently, the availability of path databases is quite limited, but there is a trend for the usage of sophisticated tracking mechanisms, such as RFID enabled tags in cars that can be read by toll ways tag readers, road sensors, GPS devices, and cameras capable of identifying cars by their license plates. Currently, we do have edge-level traffic information available, and such data can be used to mine frequent driving patterns of length one, which can be quite useful in finding fast roads (at the edge level) that are consistently taken by drivers.

Definition 8.1.4 *An edge forecast model* $F(edge\_id, t)$*, returns a tuple* $(d_1, d_2, \ldots, d_k)$ *with the expected driving conditions for edge* $edge\_id$ *at time* t*.*

The forecast model is a way for the route planner to estimate driving conditions at different edges in the graph. This is analogous, for example, to looking at the wether prediction for the day, before taking an extended trip to

plan the route accordingly. An example of the forecast function may be: "*At 5 pm* [*time*]*, for highway 57 between Champaign and Normal* [*edge*]*, Weather = rain, and Construction = no* [*conditions*]".

With the above definitions available, we are ready to state our problem:

Problem Statement. *Given a road network $G(V, E)$, a set of speed patterns S, an edge forecast model F, and a query $q \leftarrow (s, e, start\_time)$, compute the fastest route q_r between nodes s and e starting from s at time $start\_time$, such that q_r contains as many frequent driving patterns as possible.*

We can see from the problem definition that we are interested in finding fast paths, that are aware of the expected driving conditions for the trip, and that give preference to routes that are historically preferred by drivers. This problem definition encompasses factors beyond what traditional research on fastest path computation has considered. And we believe these factors can be essential in selecting desirable deriving routes in addition to being fast.

Figure 8.1: San Joaquin road network

8.2 Traffic Database

A traffic tracking system can generate information on the speed conditions for different times of day for each road in the network, such information can be represented as a set of traffic observations of the form $(edge\_id, time, speed)$, where $edge\_id$ is an edge, $time$ is the time when the observation took place, and $speed$ is the observed speed. A

car_id	path $\langle (edge\_id, start\_time, end\_time), \dots \rangle$
1	$(e_1, 10, 15)(e_2, 15, 23)(e_7, 23, 29)$
2	$(e_3, 20, 29)(e_1, 29, 33)$
3	$(e_1, 9, 16)(e_2, 16, 22)$
4	$(e_9, 10, 11)(e_2, 11, 17)(e_8, 17, 20)$
...	...

Table 8.2: Traffic database

more sophisticated system, such as the one used to monitor the San Francisco Bay Area traffic conditions [2], can use radio-frequency tags placed in each car to track the paths traversed by individual vehicles. These tags can be the same ones used for automated toll collection, the city would just install readers at many non-toll roads. In this case, each traffic observation will be of the form $(car\_id, edge\_id, time, speed)$, where car_id is a vehicle identifier, and other values are defined as before. Vehicle-level observations can be sorted on car_id and t to generate a path database, where each entry is the sequence of edges traversed by a car during a driving session. We can use either form of data to determine frequent driving patterns. If only edge-level data is available, we can use the number of edge observations as support, but in this case only frequent length-1 patterns can be mined. If we have vehicle-level data, we can mine longer frequent driving patterns.

In addition to speed information at each edge, we can augment the traffic database with the set of driving conditions present during each edge observation. Given a set of available driving factors $D_1, \dots, D_n$, we can augment each traffic observation with the tuple $(d_1, \dots, d_n)$ where each d_i is a value for driving factor D_i. Table 8.2 presents an example traffic database in path format, where each path stage is of the form $(edge\_id, start\_time, end\_time)$, where $start\_time$ is the time when the car entered the edge, and end_time is the time when the car exited the edge. For lack of space we do not show observed driving conditions at path stages.

In the above example we have that $support(e_1) = 3$, $support(e_2) = 3$, $support(e_7) = 1$, etc.

8.3 Road Network Partitioning

8.3.1 Road Hierarchy

Road networks are organized around a well-defined hierarchy of roads. An example of a typical road hierarchy is the road network of the United States, where highways connect multiple large regions, interstate roads connect locations within a region, multi-lane roads connect city areas, and small roads reach into individual houses. Information on road category is available for both the Unites States where roads are classified into 4 levels, and for the European Union where roads are classified at a higher level of detail into 13 levels. For our San Joaquin example in Figure 8.1, we

[2] http://511.org

draw roads at the two highest levels in bold, and all other roads in gray.

Most existing work on hierarchical shortest path algorithms assume that a partition of the road network is provided, or that the partition can be generated by imposing a fixed grid over the network [61, 62]. Other approaches such as [80] use the idea of Highways to divide the graph, but their definition of Highway is graph theoretic, designed to preserve optimality of routes, and does not necessarily match the road size classification. We believe that the natural partition induced by the road hierarchy itself can be used to divide the network into semantically meaningful areas, with well defined driving and speed patterns, and that can provide a significant speedup for fastest path queries when compared to arbitrary partitioning methods such as the grid-based one.

The partitioning process can first use the highest-level roads to divide the road network into large regions enclosed by such roads. Each large region can in turn be further subdivided by using the next lower-level roads. This process can be recursively applied until an area contains only roads at the lowest level of the hierarchy, or until a threshold on the minimum number of nodes in an area is reached.

Definition 8.3.1 *Given a road network $G(V, E)$, with predefined edges classes $class(e)$ for each edge e, the class of a node n denoted $class(n)$, is defined as the biggest (lowest class number) of any incoming or outgoing edge to/from n.*

The definition indicates that for an intersection of two highway segments with a small road (*i.e.*, the entry point into the highway), the intersection will be at the level of the highway, not at the level of the small incoming road.

Definition 8.3.2 *Given edges of class k, a partition $P(k)$ of a road network $G(V, E)$ divides nodes into areas $V_1^k, \ldots, V_n^k$, with $V = \bigcup_i V_i^k$. Areas are defined as all sets of strongly connected components after the removal of nodes of class k or higher from G. A node n, with $class(n) > k$ in strongly connected component i, belongs to area V_i^k, and it is said to be interior to the area. A node n, with $class(n) \leq k$ belongs to all areas V_i^k such that there is an edge e, with $class(e) < k$, connecting n to n' and $n' \in V_i^k$, such nodes are said to be border nodes of all the areas they connect to.*

Given a road hierarchy with l levels, we can construct a hierarchy of areas as a tree of depth $l - 1$, the root node represents the entire road network, children of the root node represent the areas formed by partitioning the root using level-1 edges, the nodes in each area form a graph themselves, and all the edges from E connecting two nodes in an area, are said to belong to the area. A node at level-k results from the partition of its parent node using edges of class $k - 1$. Notice that according to this convention, road class 1 is the largest, and road class l the smallest.

Figure 8.2 presents the partition of the road network for San Joaquin, CA. We have numbered each large area with two numbers $a : b$, a is the area number when roads of level 1 are used, and b is the subarea of a when roads of level 2 are used to subdivide a. We can see in the upper left corner that we have not marked individual areas, the reason is that there are quite a few strongly connected components in this region, and each has its own area, *i.e.*, nodes inside each

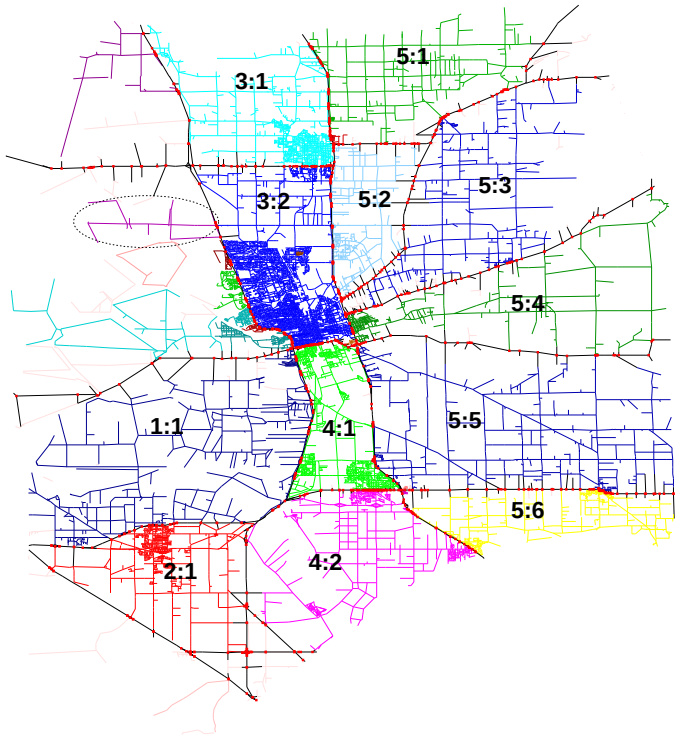


Figure 8.2: San Joaquin partitioned road network

component cannot reach nodes in other components except by going through border nodes. We have marked one such component in the figure by encircling it with a dotted line to illustrate the point. Border nodes are marked with small rectangles, which can be seen along border roads (roads at levels 1 and 2). As we can see the partition is quite natural. We also experimented with partitioning the entire road network for the San Francisco Bay Area, and for the complete state of Illinois among others (Figures not included for lack of space), and in every case the hierarchical partitioning algorithm found balanced partitions, with may areas in dense regions of the graph, and less but larger areas at more sparse regions of the graph.

8.3.2 Area Partitioning Algorithm

In this section we develop an efficient algorithm that can automatically generate a semantically meaningful partition of the road network by using road hierarchy information. The algorithm uses a flood filling technique to identify strongly connected components delimited by high level edges. We take as input the road network G and the edge class k used for partitioning. Fist we assign to each node an empty set of areas. We then choose a node n with $class(n) > k$ (*i.e.*, connected to edges less important than the ones used for partitioning), and add area a to this node's area set; at this point we move to all neighbors of the node that are reachable through edges of class greater than k, and add a to

their area sets. This process repeats until no further nodes can be reached. We are basically walking in every possible direction from the node until we reach large roads used for partitioning and we stop. At this point we increase our area number, move to the next node with an empty area set, and repeat the process until all nodes are assigned to at least one area. One nice feature of the algorithm is that it automatically identifies *interior nodes* (those with a single area in their area set), and *border nodes* (those with multiple areas in their area set).

Analysis. Algorithm 7 examines each interior node $O(1)$ times, and border nodes $O(|a|)$, where $|a|$ is the number of areas. So the order of the overall algorithm is $O(n \times a)$. In general $|a| \ll n$ the algorithm can be considered linear in the number of nodes. In our experiments we partitioned real road network graphs with more than a million nodes in just a few seconds.

Algorithm 7 Area partitioning

Input: $G(V, E)$, k edge class used to partition G.
Output: Partition $P(k) = V_1^k, V_2^k, \ldots, V_n^k$
Method:

```
1:  area = 0;
2:  Areas[n_i] = φ for all node n_i;
3:  for Each node n_i do
4:     if Area[n_i] = φ and class(n_i) > k then
5:        q.push(n_i);
6:        while not q.empty() do
7:           n ← q.pop();
8:           Area[n] = Area[n] ⋃ {area};
9:           push into q all neighbors of n reachable through edges of class greater than k;
10:       end while
11:       area = area + 1;
12:       q.clear();
13:    end if
14: end for
15: for each area i construct V_i^k as the set of nodes n such that i ∈ Area[n]
```

8.4 Traffic Mining

An important contribution of our work is to take into consideration the factors that affect driving speed as well as driving behavior in the adaptive fastest path computation algorithm. We think that route planning software has to account for all such factors in order to provide routes that are not only fast and relevant given the conditions encountered by the driver, but also well supported, in the sense that many drivers in the past have opted for such route under similar circumstances.

8.4.1 Speed Pattern Mining

In any road network multiple factors influence the speed at which we can travel through different roads. Weather, time of day, vehicle class, and road construction are just a few among the many dimensions that can influence road speed. In this section we will develop a method that mines, from a large database, a set of concise rules that identify the most relevant factors influencing road speed.

As we mentioned before, the traffic database augmented with extra factors, contains a collection of traffic tuples of the form $\langle edge\_id, time, (d_1, \ldots, d_k) : speed \rangle$ (if we have vehicle level traffic data, we compute the average speed for the edge under every condition for all cars). We can see the problem of speed pattern mining, as a classification problem where we would like to predict edge speed based on time and feature values $d_1, \ldots, d_k$. We can derive rules such as "if area = a_1 and weather = icy and time = $rush\ hour$ then speed = $1/4 \times base\ speed$". Looking at this rule we notice a few things. First, feature values reside at different levels of abstraction, *i.e.*, each factor has an associated concept hierarchy and the rules can use values at any level. Second, speed has been expressed as a relative value with respect to a base speed, such transformation allow us to build more general rules; in the above example "$1/4 \times base\ speed$" indicates that no matter the initial speed of the road, it slows down to a quarter of its initial value. Third, in this case the class label which is speed, or more concretely, speed factor, is a continuous attribute that requires some form of discretization in order to perform rule induction.

There are several different ways to do rule induction, in this chapter we chose the method of decision tree induction [76] as it provides rule predicates of the desired form, which in general have good accuracy and generality, and the method can be applied to very large datasets efficiently [38]. Before running a decision tree algorithm we run a preprocessing step to discretize speed factors, which will be treated as our class label. Speed factors can be discretized through clustering [54], and cluster centroids $\{c_1, c_2, \ldots\}$ assigned to each traffic tuple. This is beneficial for route planning application as it allows us to derive speed rules that are supported by a large number of observations and are thus statistically significant. For the time dimension, we can use its concept hierarchy to register values at a higher level of abstraction than that of the raw data, *e.g.*, from seconds to minutes. We can then treat time at the minute level as a continuous attribute which can be handled by decision tree algorithms through binary splits [77], or multi-interval discretization methods [26].

8.4.2 Driving Pattern Mining

One of the most common ways that drivers use to determine good driving routes in an unfamiliar area is to ask local people for tips, we may find for example that route R_1 is very good in the summer but that in winter the roads become unsafe, or that route R_2 although fast, goes through a high crime area and should be avoided at night. This is valuable information that has been largely ignored by route planning algorithms, and that cannot be derived by looking at just

distance and edge speeds.

Driving patterns can be derived from the traffic database by using frequent pattern mining [4, 49, 74]. We can define a minimum support level, and go through the traffic database identifying frequent edges, and if we have access to individual vehicle data, longer frequent path segments can be mined. The problem with this approach is that a uniform minimum support level is difficult to define, and it may filter many important local roads, or may keep infrequently traveled high-level roads.

We propose a frequent pattern mining method guided by the area and road hierarchies: Frequent edges are mined at the area level, using a minimum support relative to the traffic volume of each edge class in the area. This will allow us, for example, to distinguish support for edges at different levels of the road hierarchy.

With driving pattern mining, each edge (or path segment) in the road network can be marked as frequent or infrequent given a time interval and a set of driving conditions $(d_1, ..., d_k)$. The path-finding algorithm will use this information to guide the search mostly through frequent edges, and only expand infrequent ones when absolutely necessary (*e.g.*, usually at the start or end of a road when we may need to go into nearby neighborhoods).

8.5 Pre-computation and Upgrades

In this section we will present two techniques aimed at improving the performance both in terms of run time and path accuracy of fastest path algorithms by the utilization of two techniques. The first is area level pre-computation of stable paths in order to improve efficiency. The second is to upgrade certain small but very fast roads to a higher level in the road hierarchy in order to improve the accuracy (path duration vs. best possible path duration) of the algorithm.

8.5.1 Area Level Pre-computation

Many fastest path algorithms rely on pre-computing a small set of fastest paths in order to improve performance [62, 23, 80]. At one end of the spectrum we can use algorithms such as Floyd Warshall [32] to pre-compute the shortest path between every pair of nodes. In this case fastest path queries can be answered in $O(1)$ lookups, but we need $O(n^2)$ space for storing those pairs, and $O(n^3)$ time for the initial computation. At the other end of the spectrum we can do no pre-computation and dynamically find the shortest path using an algorithm such as $A*$ [53]. In between these two extremes we have several algorithms that do hierarchical decomposition of the original graph, and pre-compute a subset of paths that are helpful in connecting different areas in the graph [62, 23].

Most methods that do fastest path pre-computation assume that there is a unique path (or several but equally attractive, and thus undistinguishable) between any two nodes. When edge speed is a function of factors such as time, weather, or road conditions, the fastest path between two nodes may be different for different times and conditions,

e.g., it may differ when we leave at 8:00 am than at 10:00 am, or if we have to drive through icy roads or dry roads. In the presence of variable edge speeds pre-computed fastest paths need to be annotated with the set of conditions under which they are valid.

Definition 8.5.1 *We say that fastest path p between nodes (s, e) is conditionally stable for starting time interval t_1, t_2 given condition $c = (d_1, \ldots, d_n)$, where each d_i is a value for factor D_i or the special value * to indicate any value for D_i is valid, if $duration(p)$ is minimal among all possible paths between (s, e), when the starting time is between t_1 and t_2, and when condition $(d_1, \ldots, d_n)$ is forecasted for all edges along paths connecting s and e.*

The benefit of pre-computing a path between two nodes is proportional to the number of path queries for which the path can be used to speedup the fastest path algorithm. We can check two conditions to determine benefit. First, how many fastest path queries will go through nodes of the pre-computed path. For example, pre-computing the fastest path between two houses in an area has little value, as it will likely help a single query, while pre-computing a path between two important intersections may benefit many queries. Second, how stable is the path, *i.e.*, for how long, and for how many speed patterns is the path a fastest path. For example, pre-computing a path that is valid only between 10:07am and 10:11am has less value than pre-computing a path that applies for the entire rush hour interval, 8:00am - 10:00am. A sensible strategy is thus to pre-compute high benefit value paths.

A naive method to determine what paths to pre-compute would be to list every fastest path in the road network, under every possible condition, and rank them according to benefit value. This strategy would yield optimal results but it is clearly unfeasible as the number of paths and conditions to consider is enormous. We propose an area level pre-computation strategy where we compute certain fastest paths only within the nodes inside the area. We first define a minimum level l_m in the area hierarchy at which path pre-computation will be conducted. Within each area at level l_m we choose the set of nodes S_p of class $l_m + 1$ (one level smaller than the borders of the area), and class l_m, and pre-compute fastest paths between nodes in S_p. For this step, we guide our pre-computation by the set of speed rules mined for the area, and limit the analysis paths involving edges with few speed rules. We can handle time intervals by using the algorithm presented in [63], which efficiently computes the set of fastest paths between two nodes for different time intervals.

8.5.2 Small Road Upgrades

The main assumption of hierarchical path finding algorithms is that drivers take the largest road available in order to reach their destination, and thus the search space for route finding can be significantly reduced. Our observation is that although this strategy is generally true, there is an important exception, if there is a small road that is faster than a large road, people will take it. For example, people driving to or from cities usually do it through highways, but during

rush hour highways can become so congested that taking smaller roads yields shorter travel times. If we ignore such cases we may incur in significant error.

Our strategy in dealing with this problem will be to *upgrade* certain edges inside an area if under some driving conditions they have a significantly higher speed than the edges at the area borders under the same driving conditions. For the above example, we would upgrade the internal edges in the area where the city resides, to the level of highways but only for the driving condition of rush hour. This way we can still compute most routes considering only highways, and incur the extra cost of looking at the upgraded edges only when absolutely necessary.

More formally, an edge e, residing in area a with border edges at level l of the road hierarchy, will be upgraded to level l if three conditions are met (i) the edge speed under driving conditions $(d_1, \ldots, d_n)$ for some time t is faster than the average edge speed of border edges in the area under the same conditions and time, (ii) edge e is at level $l + 1$, and (iii) the edge is frequent. For all such edges we will register a conditional road class tuple $\langle edge\_id, t, (d_1, \ldots, d_n) : upgraded\_class \rangle$.

Algorithm 8 Edge upgrading

Input: $G(V, E)$: road network, T: area hierarchy
Output: List of upgraded edges
Method:

1: Precompute the average border speed for every area under every valid driving condition and time
2: q ← push all leaf areas in T;
3: **while** q not empty **do**
4: A ← q.pop();
5: **for** each edge e in A **do**
6: **if** e.level = A.level + 1 and e is frequent **then**
7: for each driving condition c and time t for which e.speed $>$ average border speed for c in A, make e.level = a.level and output $\langle e, t, c, e.level \rangle$;
8: **end if**
9: **end for**
10: q.push(A.parent);
11: **end while**

Analysis. Algorithm 8 presents the method used to upgrade internal area edges when they are faster than the border. The algorithm starts by computing the average speed of border edges to an area for valid conditions. This can be done efficiently in a single scan of the list of edges. We then traverse the area tree in a bottom up order, upgrading edges at the lowest areas, before upgrading edges at larger areas. Notice that an edge can be upgraded multiple times if it is consistently faster than the borders of several successively larger areas. During the edge upgrade process each edge is touched $O(l)$ times where l is the number of levels in the tree, so in total we touch an order of $O(|E| \times l)$ edges.

8.6 Fastest Path Computation

In this section we will introduce an approximate fastest path algorithm for road networks, that computes fastest paths between a source and destination node, such that the computed route has the following properties:

- Fastest routes should be well supported by the historical driver behavior, *i.e.*, they should contain as many frequent driving patterns as possible.
- Fastest routes between a source and a destination will go through the largest possible roads connecting the two locations as long as there are no smaller roads along the way that have a significant advantage over the large ones.
- Fastest routes will account for all relevant factors affecting driving speed expected to occur during the trip such as weather, time of day, and road construction status.

Before running the algorithm we assume that the following components have been computed:

- The road network G has already been partitioned using algorithm 7, and we have an area hierarchy tree T that encodes the parent/child relationship between areas.
- Speed patterns have been mined and we can use the function $get\_edge\_speed\ (edge\_id,t,(d_1,\ldots,d_k))$ to get the speed of $edge\_id$ when it is taken at time t, and for driving conditions $(d_1,\ldots,d_k)$. This information is retrieved from our mined set of rules for driving conditions, by selecting most specific rule(s) applicable given the conditions.
- Driving patterns have been mined and we can use the function $is\_frequent(edge\_seq, t, (d_1,\ldots,d_n))$ to determine if the edge sequence $edge\_seq$ is frequent under conditions $(d_1,\ldots,d_n)$ at time t.
- We have pre-computed a set of area-level fastest paths with high benefit value.
- We have used algorithm 8 to upgrade internal roads to an area when they are faster than roads along the area border for a given set of time and driving conditions. We can retrieve upgraded edges with the function $get\_edge\_class\ (edge\_id, t, (d_1,\ldots,d_k))$ that returns the class of $edge\_id$ for driving conditions $(d_1,\ldots,d_k)$ at time t.

8.6.1 Algorithm

At this point we are ready to state our fastest path algorithm, it is a variation of $A*$, where we dynamically compute edge costs, take advantage of pre-computed paths, follow edges in ascending/descending order of their level in the road hierarchy, and give priority to frequent edges (or edge sequences).

The key technical contributions of the algorithm are three. First, it incorporates previously neglected factors such as speed and driving patterns into route finding. Second, we improve performance by utilizing a novel area level pre-computation scheme. And third, although hierarchical path finding has been used in the past, to the best of our knowledge this is the first study that uses the idea of small road upgrading to improve path quality with minor impact to efficiency.

The key concepts of the algorithm are summarized below:

1. We maintain a priority queue of expanded paths (represented by the last node of the path), for each path we keep $g(n)$ *the current cost* (*travel time*) *spent to get from start to* n, and $h(n)$ *the expected cost to reach the end node from* n.
2. At each step of the search process we pick the node with lowest $g(n) + h(n)$ value that is frequent[3], if no frequent node is present in the queue we pick the best infrequent one. We prefer to travel through frequent roads, but sometimes it is necessary to pick a small number of infrequent paths in order to reach the destination. This can be specially true around the starting and ending nodes.
3. At the beginning of the search we determine, using the area hierarchy tree T, what is the lowest common ancestors of both start and ending nodes. We use the lowest common ancestor to set the phase of the algorithm, which can be *Ascending* or *Descending*. We are in an *Ascending* phase when the currently examined node is in an area that is below or at same level of the lowest common ancestor, and we are in a *Descending* phase otherwise.
4. At each iteration we look at the neighbors of the node currently being examined. If we are ascending, we only consider neighbors that are connected through edges that have an edge class lower or equal to the previous edge (bigger road) in the path. If we are descending, we only take edges with class greater or equal to the previous edge (smaller road) in the path. The class of each edge is dynamically computed by calling the function $get\_edge\_class\ (edge\_id, t, F(edge\_id, t))$, where F is the predictor function that returns the tuple $(d_1, \ldots, d_k)$ of expected driving conditions (*e.g.*, weather, road construction, etc) for the edge at time t, t is the projected time at which the edge will be taken. The set of neighbors of a node are all those nodes directly reachable through a single edge, or indirectly reachable through a pre-computed path.
5. Whenever we insert a new path into the priority queue, we update its $g(n)$ value by adding to the path's total travel time the time to traverse the new edge, which is computed by dividing the edge distance by the expected edge speed retrieved with the function
   $get\_edge\_speed\ (edge\_id, t, F(edge\_id, t))$. $h(n)$ is also updated for the path by using a conservative estimate of the total travel time from the current node to the goal node. Several different estimation policies are possible,

[3] A more sophisticated strategy is possible, we can give preference to paths that have longer frequent driving patterns, than shorter ones.

more accurate ones can significantly speed up the search [24]. In our implementation we used the simple heuristic $h(n) = distance(n, end)/max\_speed$, but any other heuristic could be plugged into the algorithm.

Lemma 8.6.1 *The adaptive fastest path algorithm, when computing a path between $(start, end)$ nodes, in areas a_i, a_j respectively will consider at most $O(|a_i| + |a_j| + |bn| + |un|)$ distinct nodes, where $|a_i|$ is the number of nodes in area a_i, $|a_j|$ is the number of nodes in area a_j, $|bn|$ is the total number of border nodes in all areas, and $|un|$ is the number of nodes connected to upgraded edges in all areas.*

Proof Sketch. The worst case for path finding is when no pre-computed path is available. In this case we need to examine all nodes in the starting area until border nodes are reached, and all nodes in the ending area until we reach the destination. Once we have reached the border nodes of the first area, the algorithm only goes through border nodes, or upgraded nodes until the destination area is reached. ∎

The implication of lemma 8.6.1 is that our search algorithm will need to consider significantly fewer nodes than traditional $A*$ even in the case when no area pre-computation is possible. We verify this result running the search algorithm on real road networks for the United States where we observed an order of magnitude savings in the number of nodes examined by the algorithm.

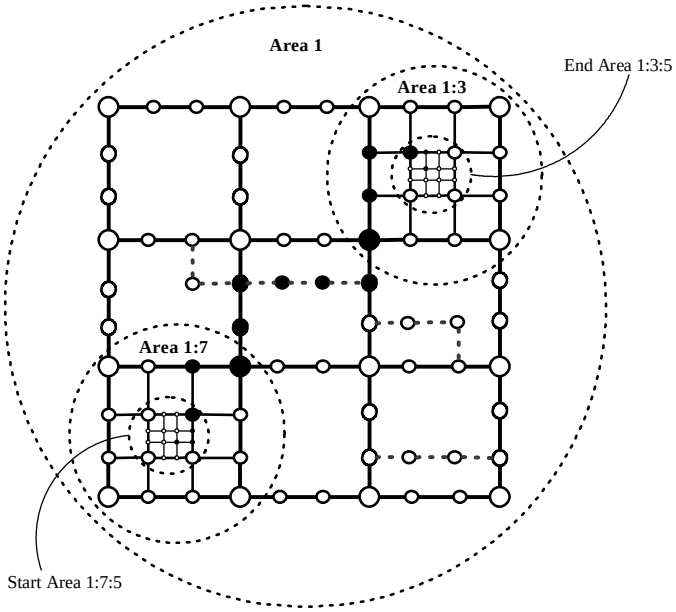


Figure 8.3: Example hierarchical search

Example (Search). Figure 8.3 presents a road network, there are 3 levels of roads, and we have partitioned the graph along the first 2 levels. The first level gives us the large grid, and the second level the finer grid (shown only for two areas). The size of nodes indicate the level at which they are, larger nodes are associated with edges at higher level.

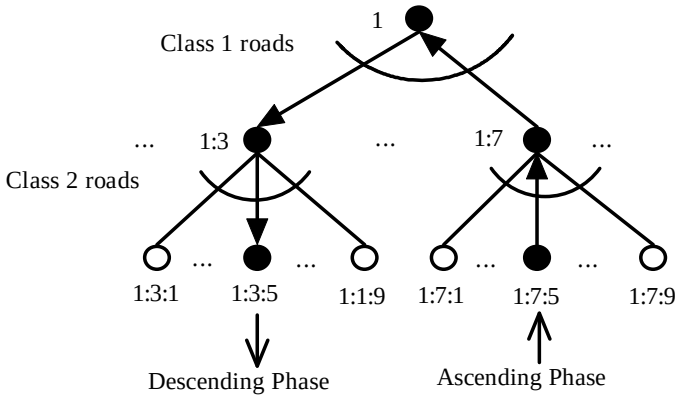


Figure 8.4: Example area hierarchy

The graph also shows edges that have been upgraded by painting them with dotted lines. The name of an area indicates its position in the area tree, *i.e.*, area $1:7:5$ is a child or area $1:7$ which in turn is a child of area 1 (more clearly seen in Figure 8.4). The total number of nodes in this graph is $28 \times 28 = 784$.

We would like to find the fastest path between a starting node in area $1:7:5$, and an ending node in area $1:3:5$. Figure 8.4 presents the area hierarchy, in this case see that the lowest common ancestor of the start and end nodes is area 1. The algorithm will proceed in two phases: first ascends from area $1:7:5$ to area 1, and then descends to area $1:3:5$. We first expand nodes in area $1:7:5$ by following roads of level 3, until we reach edges in the border of the area which are at level 2, and are inside area $1:7$. At this point we are still ascending, as we have not reached the lowest common ancestor, and we will not look at roads of level 3 any more. We now move along edges of level 2 until we reach the borders of area $1:7$, which are roads of level 1. At this point we will consider only roads of level 1, and roads of level 2 upgraded to level 1 (the dotted ones) until we reach area $1:3$, at which point we enter the descending phase of the algorithm and the order of road classes to follow is reversed. We follow roads of level 2 to area $1:3:5$, and finally roads of level 3 to the destination node.

In this example, in order to simplify the explanation we have ignored edge frequency. But if we consider that all edges of level 3 are infrequent, it should be clear that when we start the search, and when we end the search, the priority queue will contain only nodes that are reached through an infrequent edge and thus we would have to use it. We have also assumed that there are no pre-computed paths; otherwise, the search process is the same, but instead of following direct neighbors of the node we also follow indirect neighbors connected through pre-computed paths.

In this example the maximum number of distinct nodes that the algorithm would have to consider is 32 at the start and end areas, 32 at the level 2 areas, 64 at level 1, and 7 nodes along upgraded edges. This is much smaller than the potential nodes examined by A* which in the worst case is the entire network, 784 nodes.

8.6.2 Online Path Re-computation

When we compute a fastest route, the predictor function F is used to estimate driving conditions throughout the entire trip, but it is possible that our initial estimate is wrong. For example, if we computed the route using a prediction of good weather for the duration of the trip, but on the half way into it a snow storm happens, our fastest path may be far from optimal. Another example of changing conditions would be an unexpected road closure or an accident. If we plan a complete trip before the starting time using a static model, there is not much that the system can do about these unexpected cases (*e.g.*, when we ask a site such as MapQuest to give us a route before the trip starts). But if we are operating in an online navigation environment, we can do better, as it is possible to recompute the best route as driving conditions change. In this case we can just apply the algorithm presented in the previous section with a starting node changed to the current position, starting time to the current time, and an updated forecast model.

In an online navigation system, having access to an *efficient* route computation algorithm is even more critical than in the static route computation model, and the power of our proposed method would be more evident. Another possible change to our algorithm, when forecast is updated, would be to invalidate our speed model for edges near the vehicle's current location, as we may know the exact speed for those, and use mined speed patterns only for farther away edges.

8.7 Experimental Evaluation

In this section we perform a thorough analysis of the adaptive fastest path finding algorithm proposed in the chapter, and compare its performance against a basic A* implementation, and a version of the adaptive of our own algorithm that does not perform area pre-computation. All the experiments were conducted on a single core of an AMD Athlon 64 X2 Dual Core processor with 2GB of RAM. The system ran cygwin and gcc 3.4.4.

8.7.1 Data Synthesis

In all of our experiments we used real road maps from areas of the United States. We used three maps. San Francisco Bay area with 175,343 nodes, and 223,606 edges. A map for the entire state of Illinois with 831,524 nodes, and 1,048,080 edges. A smaller map for San Joaquin, CA with 18,496 nodes, and 24,123 edges. We simulated different traffic conditions using the Network-based Generator of Moving Objects by Thomas Brinkhoff [11], which is a well known traffic simulator. For each map we ran two simulations, one with 10,000 objects used to generate rush hour like speed conditions, and one with 1,000 objects used to simulate non-rush hour speed conditions. In each simulation we defined two object classes, cars with faster speeds, and trucks with slower speeds. We also incorporated a weather factor into the simulation by running the simulation with and without external objects, which in the data generator can

be used to slow down certain areas of the road network as if bad weather were occurring. The output of the simulation was a list of edge observations of the form $\langle edge\_id, car\_id, time, weather, speed \rangle$. The output files were a few hundred megabytes long. Using this information we mine speed patterns for each edge, such patterns involve the dimensions of $time$, $weather$, and $vehicle\_type$.

In most of the experiments we compare three methods. The first is an implementation of $A*$, we run this algorithm on the entire road network. We maintain a priority queue of paths to expand. At each iteration we select the path with minimal $g(n) + h(n)$, where $g(n)$ is the current cost of the path, and $h(n)$ is the expected cost to the goal. For each path we update $g(n)$ by retrieving the appropriate speed for the edge at the time when it will be taken, and for the type of vehicle for which the route is being planned, and for the forecasted weather. $A*$ always finds the fastest possible path, and is thus our baseline for correctness. The second algorithm $Hier$ is our adaptive fastest path algorithm implemented without area pre-computation, *i.e.*, no fastest paths have been pre-computed at all, and without considering road upgrades, *i.e.*, small roads through an area that are faster than roads in the area border. The third algorithm, $Adapt$ is the fastest path algorithm proposed in this chapter.

8.7.2 Query Length

In this set of experiments we vary the distance between the starting node and the ending node of the query. For this experiment we used the San Francisco road network. The road network was partitioned with edges of level 1, and level 2. We artificially upgraded a single random path in 20% of the lowest level areas, and set the speed for edges in upgraded paths higher than the average speed of the border edges of the area. We pre-compute fastest paths in 30% of the lowest level areas. Results are the average of 100 random queries. We varied the average distance between the starting and ending nodes. The longer the distance the larger the search space. Query Length measures the distance of the end points in the query as a percentage of the map diameter.

In Figure 8.5, we see the number of expanded nodes for the three algorithms. We see that $A*$ the number of nodes expanded by $A*$ grows very rapidly for large paths. This is expected as there are many more possible paths to consider between nodes that are separated by a large number of edges. At the same time we see that $Hier$ and $Adapt$ expand a number of paths that is almost constant. The reason is that both algorithms limit their search to larger roads and only go into individual areas at the start or end of the search. We see that although $Adapt$ has to consider all upgraded edges, it only expands slightly more nodes than $Hier$ which ignores those edges.

Figure 8.6 presents the travel time for the three methods. We see that $A*$ always gives us the fastest path. The path found by $Adapt$ is almost as good as the $A*$ path, but at only a fraction of the cost. $Hier$ suffers significantly in terms of path travel time, the reason is that it completely ignores roads internal to areas that are faster than border roads. This experiments shows the power of our algorithm, not only in terms of efficiency, but also accuracy, and highlights

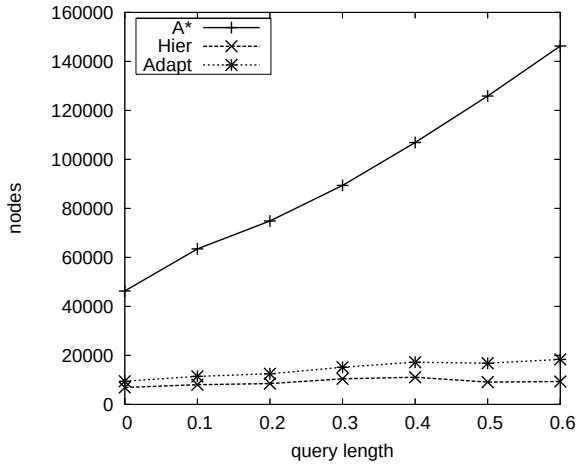


Figure 8.5: Query length vs. Expanded nodes. Depth: 2

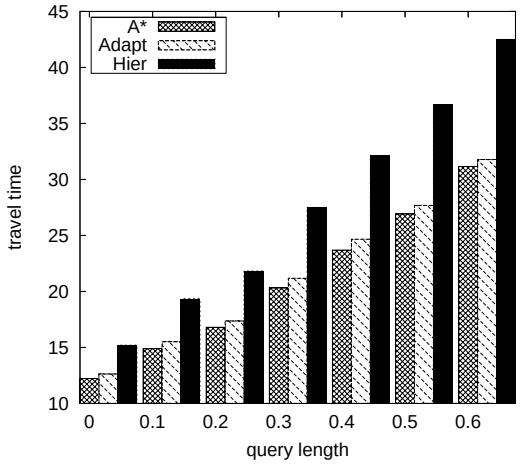


Figure 8.6: Query length vs. Travel time. Depth: 2

the relevance of considering small fast roads in addition to large roads.

Figure 8.7, presents the average CPU time per query using the 3 algorithms, again we observe the same pattern as in the expanded nodes figure.

8.7.3 Upgraded Paths

In this set of experiments we vary the percentage of lowest level areas that contain an upgraded path. For this experiment we used the San Francisco road network. Average path length is 50% of the area diameter, and all the conditions are the same as the ones used for the query length experiments, except for upgraded paths which we vary.

We can see in Figures 8.8 and 8.9 that neither $A*$ or $Hier$ are affected in terms of the number of expanded nodes,

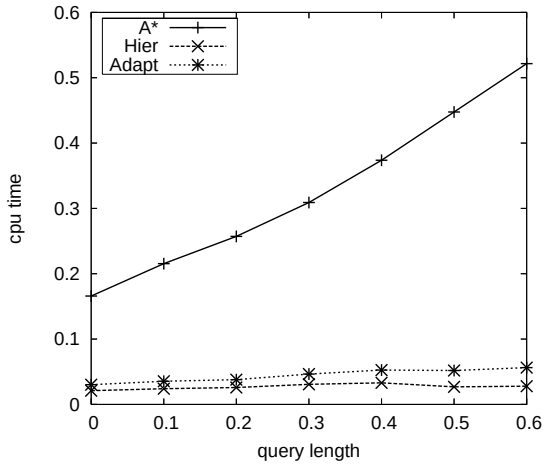


Figure 8.7: Query length vs. CPU time. Depth: 2

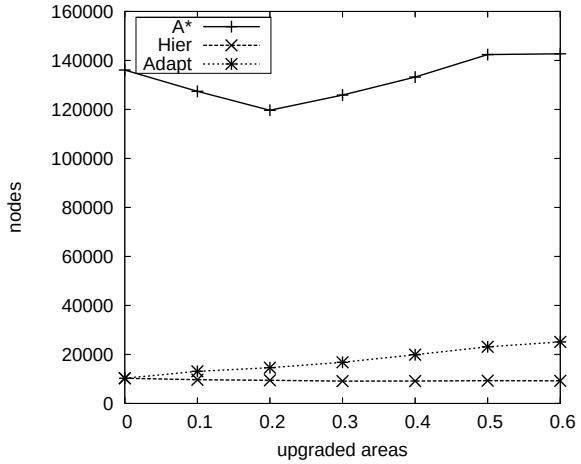


Figure 8.8: Upgraded paths vs. Expanded Nodes. Depth: 2

or the CPU speed. This is because $A*$ always considers the entire network, and $Hier$ disregards upgraded edges completely. The performance of $Adapt$ suffers as we have more upgraded edges that need to be considered in the search process. But we can see that the degrade in performance is quite gradual we go from around an average of 18,000 expanded nodes when no edges are upgraded to just around 20,000 when 60% of the areas contain an upgraded path.

Figure 8.10, really highlights the important of path upgrading, we see that when no edges are upgraded both $Hier$ and $Adapt$ perform equally, as we increase the number of upgraded edges $Adapt$ starts closing the gap with $A*$, this is expected as we have ever more options to find a good path, while the quality of paths found by $Hier$ continues to decrease. This experiment is important because we see that we can use a fairly aggressive edge update strategy to

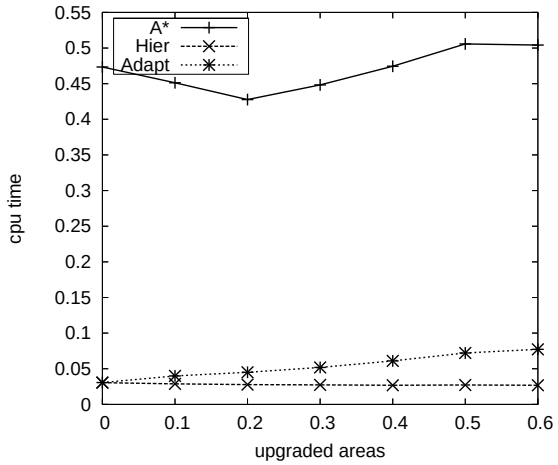


Figure 8.9: Upgraded paths vs. CPU time. Depth: 2

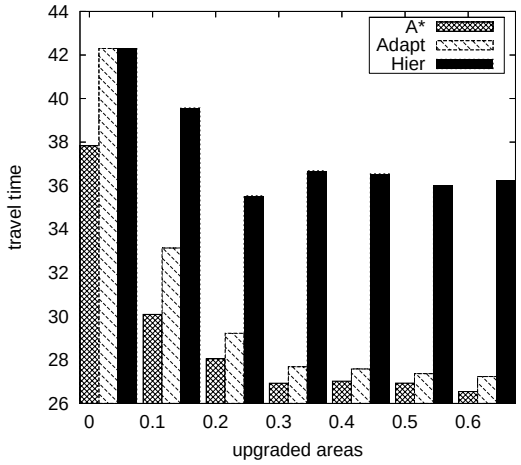


Figure 8.10: Upgraded paths vs. Travel time. Depth: 2

improve path quality without incurring in a significant performance penalty. We could consider, for example, interior edges as long as they are 80% as fast as border edges to improve path quality.

8.7.4 Area Pre-computation

In this experiment we examine the performance gain for different levels of pre-computation. We use the San Francisco road network, partitioned using roads of classes 1 and 2. The average path length used is 50% of the area diameter. The percentage of areas with an upgraded path is 20%. We compare two methods, $Adapt$ which is our algorithm, and $Adapt_n opre$ which is the same algorithm but without using pre-computed areas. For this experiment we select a percentage of the lowest level areas, and pre-compute every fastest path in it. We vary the number of pre-computed

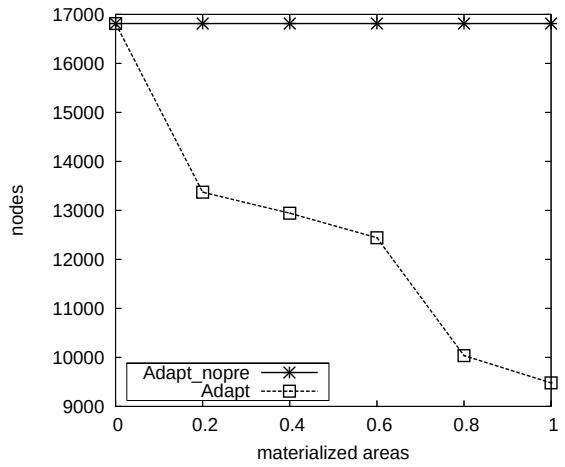


Figure 8.11: Pre-computed areas vs Expanded nodes. Depth: 2

areas at the lowest level from 0% to 100%. We can see that the performance improvement is very significant as we can use more pre-computed areas, *i.e.*, the algorithm is basically reaching for the destination taking very long jumps. In this experiment we did not pre-compute any higher level area, if we had done it, the performance gains would have been even more noticeable.

8.7.5 Road Network Size

In this experiment we compare query processing efficiency for 3 road networks of different sizes: San Joaquin (sj) with 18,496 nodes and 24,123 edges, San Francisco (sf) with 175,343 nodes and 223,606 edges, and Illinois (il) with 831,524 nodes and 1,048,080 edges. All the maps where partitioned using the top two levels of roads. Average path length was 50% of map diameter. 20% of areas had a single path upgraded.

We can see in Figure 8.12 that the adaptive algorithm has excellent scalability in terms of road network size. The reason is that the number of nodes usually growth much slower than the number of small roads, and thus our algorithm is able to significantly restrict the search space to a manageable size even in the presence of millions of nodes and edges. This result is very encouraging, as it indicates that the algorithm can handle route planning in very large maps very quickly, a factor that is essential when we consider that the number of queries that such system needs to handle is in the billions.

8.8 Summary

In this chapter we developed an adaptive fastest path algorithm, that bases routing decision on driving and speed patterns mined from historical data. This is a radical departure from traditional algorithms that have focused only on

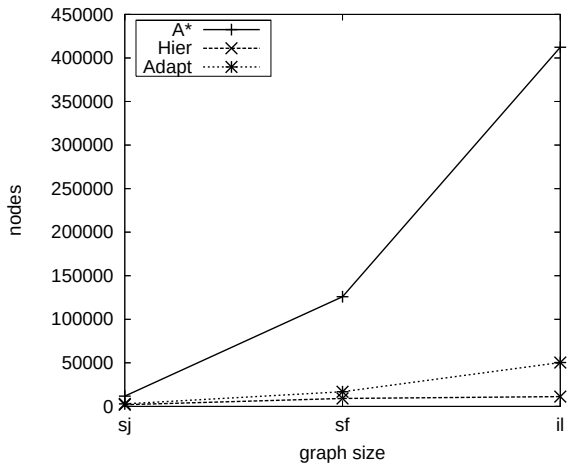


Figure 8.12: Graph size vs. Expanded nodes. Depth: 2

speed and Euclidean distance considerations. The routes computed by our algorithm are not only fast given a set of driving conditions but they also reflect observed driving preferences. This is in sharp contrast to existing algorithms that may send a driver through high crime areas of a city at night, or through unsafe roads in order to save a few minutes of travel time.

A road network partitioning algorithm was introduced. The algorithm uses the hierarchy of roads to segment the network into areas that are enclosed by large roads. This method yields very natural partitions, where large areas are observed at regions with low road densities, and much finer areas are observed at dense regions such as big cities. Areas are a central concept to route planning, as they provide the basis for hierarchical path finding, area level pre-computation, and area sensitive support of driving patterns.

We showed that very significant query processing gains can be obtained, by following the principle that drivers tend to travel through the largest roads available for the trip, unless small road along the way have a speed advantage over the large ones. We also presented a method to identify such fast roads, and efficiently incorporate them into route planning. In our experimental study we demonstrated that incorporating fast small roads into the hierarchical path finding algorithm can significantly improve the quality of routes.

Through an empirical study, on real road networks, using a realistic traffic information, we verify the very large performance gains of our algorithms vs. competing methods, while showing that computed routes are very close to optimal.

Chapter 9

Mining Traffic Anomalies

Identification and characterization of traffic anomalies on massive road networks is a vital component of traffic monitoring [44]. Anomaly identification can be used to reduce congestion, increase safety, and provide transportation engineers with better information for traffic forecasting and road network design. However, due to the size, complexity and dynamics of such transportation networks, it is challenging to automate such a process. We propose a multi-dimensional mining framework that can be used to identify a concise set of anomalies from massive traffic monitoring data, and further overlay, contrast, and explore such anomalies in multi-dimensional space.

9.1 Anomaly Mining Framework

In this section we give a brief overview of our proposed anomaly mining framework (Figure 9.1).

Traffic data acquisition. This module is in charge of collecting sensor data from different road sensors, cleaning the readings, and deriving measures not directly provided by the sensors, *e.g.*, speed from occupancy and flow. In this chapter we focus on sensor readings originating at inductive loop detectors, which are the most common type of sensor currently available, but the framework allows for the use of data from multiple sensors.

Anomaly mining. This module takes cleansed data from the data acquisition layer and runs an anomaly mining algorithm that identifies spatially and temporally connected segments that present anomalous traffic conditions. At this stage we perform significant compression (around two orders of magnitude) over the full set of single segment anomalies, and enhance the semantics of anomalies with properties such as total duration, length, or propagation speed. The compression provided at this stage is essential for the efficient implementation of anomaly overlay and anomaly cubing.

Multi-dimensional anomaly overlay. This module overlays multiple anomalies, along criteria such as topology or severity. It provides data warehousing facilities for the computation of a data cube, which provides anomaly overlay over the space of contextual dimensions, such as weather, geography, and time, and facilitates multidimensional OLAP analysis, *i.e.*, drill down, roll up, slice, and dice over anomaly clusters.

In the subsequent sections, we will explore each module in the framework in greater detail.

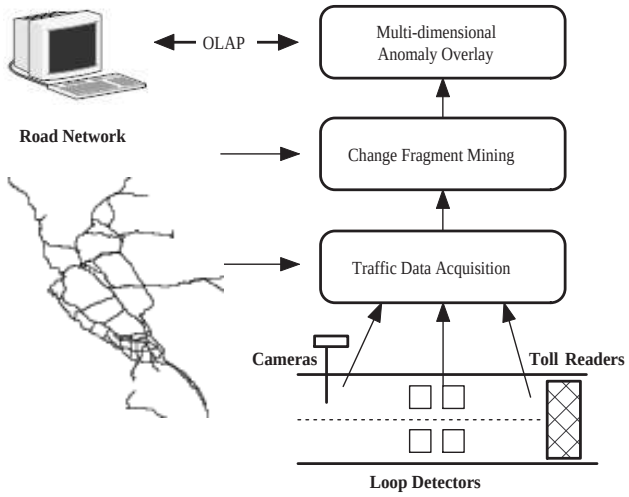


Figure 9.1: Anomaly mining framework

9.2 Traffic Data Acquisition

9.2.1 Traffic Monitoring System

Many large cities in the United States have Transportation Management Centers (TMCs) that continuously monitor the state of roads (mostly highways) through the use of sensors placed in the road network.

There are many uses for the information collected by TMCs. Example applications include, incident detection [75, 48], which tries to associate anomalous patterns with the occurrence of incidents; generation of estimates of driving times seen in signs over highways; identification of performance measures related to traffic, and helpful to transportation engineers and policy makers for making better decisions regarding to road network upgrades.

9.2.2 Data Collection

The road network in the United States is monitored by a variety of sensors, that collect real time traffic information. A few of the most common sensors are:

Inductive Loop Detectors. These are closed wire loops embedded into the pavement. When a vehicle passes on top of the detector, a current is generated and a controller by the side of the road registers the change in current in the loop. This information is transmitted to a local computer monitoring several detectors and then sent to the TMC.

Two measures can be calculated from single inductive loops: (1) *flow*, *i.e.*, the number of vehicles that went over the sensor in a time period, and (2) *occupancy*, *i.e.*, the percentage of time that the loop detector was active in a time period. Double loop detectors, which are two single loop detectors placed in close proximity, provide the *average speed* of vehicles passing through the detector in a time period.

Inductive loop detectors have been in use for a long time, but the data they generate is known to be noisy [8]. The sensors sometimes over and under count vehicles, or stop registering vehicles due to wear and tear. Any system relying on loop detector data has to perform data cleaning [18] in order to conduct reliable analysis.

Cameras. Video cameras are becoming popular in traffic monitoring and many highways, road intersections, and toll booths have cameras that record passing vehicles. Usually cameras are observed by human operators who check for interesting events. Recently, systems of automatic vehicle identification based on cameras have been developed, and as computer vision techniques improve, cameras could be used to track individual vehicles on the road network.

GPS and RFID. Today many vehicles are equipped with toll booth tags that are, not only, detected by readers placed at toll booths, but that can also be read by readers placed at various locations in the road network. 511.org[1], a traffic monitoring system in CA, tracks individual vehicles with toll booth tag readers placed at certain locations in the city. Also, vehicles equipped with GPS devices can provide a detailed record of their trajectories through the city, some commercial trucking and shipping companies collect GPS information from many of their vehicles, and this data could be used by TMCs for monitoring and planning.

Both Camera- and GPS-based sensors collect information on individual vehicles, and thus can give a more accurate picture of how traffic flows in the road network, but such data is still scarce. Loop detectors, on the other hand, only provide aggregate information of traffic at discrete points in the road, however, they are widely available, and large datasets of loop detector readings are available.

9.2.3 Road Network

Definition 9.2.1 *Road network is a graph $G(V, E)$, where V is the set of road end points and intersections and E is the set of road segments. An edge e_{ij} exists if there is a road that directly connects two nodes v_i and v_j. Each $e_i \in E$ and $v_i \in V$ has an associated set of properties describing it.*

A vertex can have spatial location, *i.e.*, latitude and longitude, as its associated property; whereas an edge can be associated with road type (*i.e.*, highway, collector, or local) and the list of sensors (with locations) placed on the road. It is common for multiple sensors to be located in a single edge of a road network, and such edges can be split into a sequence of single sensor edges. Such transformation facilitates analysis by allowing the study of traffic measures at the single edge level.

Given a road network $G(V, E)$, we can divide each edge e_{ij} between nodes n_i and n_j, according to sensors $s_1, \ldots, s_k$ located there into $e_{ij1}, \ldots, e_{ijk}$, along the path formed by the sequence of nodes $n_i\ n'_1 \ldots, n'_{k-1}\ n_j$. The graph formed by applying such procedure to each edge is itself a road network with at most one sensor per edge.

[1] http://511.org

Edges with a sensor are said to be *monitored*. The location of nodes at the end points of monitored edges is computed as the midpoint between the sensors in the edges before and after the node.

Figure 9.2 presents a portion of the highway network in the bay area in California, where road colors correspond to observed speed: green is normal, and red is slow. The Figure was taken from PeMS[2].

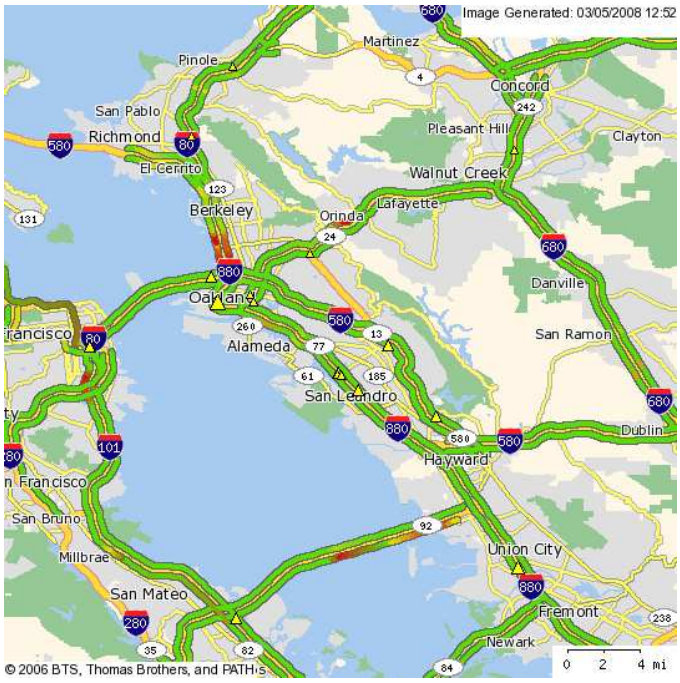


Figure 9.2: California road network

9.2.4 Sensor Data

Data collected from inductive loop detectors in the road network is of the form $\langle vds\_id, lane, time, flow, occ, speed\rangle$, where vds_id is a unique identifier for the sensor, $lane$ is the lane where the measurements were collected, $time$ is the time when the collection took place, occ is the percentage of the time that the sensor was occupied (a vehicle was on it), $flow$ is the number of vehicles that passed through the sensor, and $speed$, if available, is the average speed of the vehicles going through the sensor.

Each sensor also has an associated set of properties, $\langle vds\_id, eid, lane\_count, type\rangle$, where vds_id is defined as before, eid is the road segment where the sensor is located, $lane\_count$ is the number of lanes that are monitored in the road, and $type$ defines if the road segment is the main line of the freeway, an exit ramp, or an entry ramp.

Individual vehicle level tracking data was scarce in the past but is becoming more available as RFID and GPS devices become more common, and camera-based identification of vehicles improves. This data has the form $\langle vin,$

[2] `https://pems.eecs.berkeley.edu/`

eid, $time\rangle$, where vin is a vehicle identifier, eid is a road segment, and $time$ is the time when the vehicle passed through the segment.

At this point Transportation Management Centers have very limited access to vehicle-level data. For example, the transportation departments at California and Texas provide road-level data, but have very little or no vehicle-level data. For this reason, we will focus on anomaly detection when only road-level data is available. But, in the case that trajectory data is at our disposal, the methods developed here can be easily extended to handle it. Trajectory data can be used to develop a more precise traffic model and improve anomaly detection by having concrete data on the speed at which vehicles traverse a sequence of road segments, not just aggregated data that may hide real traffic conditions.

9.3 Anomaly Mining

9.3.1 Traffic Model

Before defining anomalous traffic, we need a model of "normal" traffic. For example, the normal speed for a highway segment in L.A. can be 35mph during rush hour, and 55mph otherwise; the same segment may exhibit slower speeds during bad weather, and faster speeds on weekends. In order to model traffic, it is necessary to account for factors that impact road speed, occupancy, and flow. Thus a traffic model should provide expected values for traffic measures, given a set of factors, such as weather or day of week.

Definition 9.3.1 *In a traffic monitoring system, with traffic factors $D_1, \ldots, D_n$, and measures $M_1, \ldots, M_k$, a traffic model is a function $f_l(eid, t, (d_1, \ldots, d_n))$ that maps traffic conditions $(d_1, \ldots, d_n)$ for road segment eid at time t into a set of expected traffic measures $m_1, \ldots, m_k$, where d_i is a value for traffic factor D_i, and m_i is a value for traffic measure M_i.*

For example, given weather and day of week as traffic factors, and speed and flow as measures, $f_l(eid, t\,(weather = snow, day = Sunday))$ could be $\langle speed = 45mph, flow = 5\ vehicles/min\rangle$.

There can be many implementations of the traffic model. A simple but effective one is to build a table of historic traffic with aggregate measure values for different traffic factors. The table is of the form $\langle eid, t, D_1, \ldots, D_n : M_1, \ldots, M_k\rangle$, where dimension and measure are defined as before, and t is time aggregated to an interesting level (*e.g.*, hour). The model can be constructed by a single scan of the traffic database, computing average measures for each combination of traffic dimension values. Alternative traffic models based on regression or decision trees [43] have been proposed and can be used by our algorithm as well.

9.3.2 Measuring Anomalies

When determining traffic change on an edge with respect to normal behavior, we have several options. For the case of inductive loop detectors, the available measures are flow, occupancy, and speed. In practice we use Occupancy as a proxy for Density.

These measures are related by the fundamental diagram of traffic flow [37] which plots the flow of vehicles for different levels of occupancy. Figure 9.3 presents an example of the fundamental diagram for a road segment in California. The slope of lines from the origin to the flow/occupany curve is the speed at which vehicles travel for the particular combination of occupancy and flow. The fundamental diagram is a fixed property of any road design and driver population. The effects of congestion on speed can be observed in the two speed lines shown, we see that as occupancy increases from 0.07 to 0.1, flow and speed decrease.

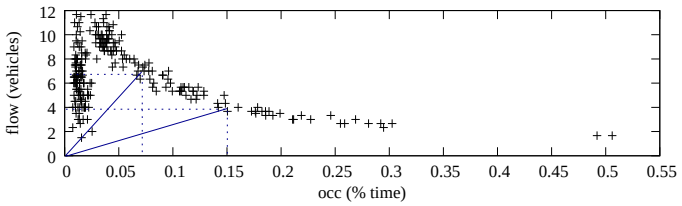


Figure 9.3: Occupancy vs. Flow

In our implementation we use speed as the basic measure for anomaly detection, because it captures the intuition of traffic anomalies as slow or fast moving traffic, and it has been widely used in incident detection. But, any other reasonable measure can be used without any change to the algorithms.

To study traffic anomaly, we introduce the concept of the *level of change* with respect to a traffic model.

Definition 9.3.2 *The* level of change $chg(e,t)$ *for edge* e *at time* t *with respect to the traffic model is defined as* $speed(e,t)/f_l(e,t,(d_1,\ldots,d_n))$*, where* $speed(e,t)$ *is the speed of edge* e *at time* t*,* f *is the traffic model, and* $(d_1,\ldots,d_n)$ *are values for traffic factors.*

Definition 9.3.3 *The speed at edge* e *at time* t*,* $speed(e,t)$*, for traffic conditions* $(d_1,\ldots,d_n)$ *is* anomalous *if* $|1 - chg(e,t)| > \delta_l$.

9.3.3 Atypical Fragments

Atypical fragment is our proposed basic unit to measure traffic anomalies. Intuitively, an atypical fragment is a set of significant changes of speed, with respect to the traffic model, at a set of edges that are spatially connected, and the intervals of change at each edge are temporally connected (*i.e.*, they occur at overlapping times).

Example. A common atypical fragment corresponds to the effects of an accident or a jam in a highway. If vehicle flow is significant, and the accident considerably reduces capacity, we should observe a decrease in the speed of edges upstream and an increase in speed of edges downstream. The changes should propagate through consecutive edges at a certain rate that depends on the reduction in capacity and the volume of vehicles on the highway. As observed by a driver, cars in the edge ahead of him/her will slow down a few seconds before he/she slows down, then both, the cars in his edge and that in the following edge will slow down, and when the accident is removed, cars in the edge ahead of him will start to speedup a few seconds before the cars in his/her edge. ■

Before we formally introduce the atypical fragment we need to provide a few definitions.

Definition 9.3.4 *When $chg(e,t)$ is anomalous for all $t \in [t1,t2]$, but normal for $chg(e,t1-1)$ and $chg(e,t2+1)$, we say that $c = \langle e, t1, t2\rangle$ is an* atypical interval *for edge e.*

Definition 9.3.5 *The* level of change *$lvl(c)$ for an interval $c = \langle e, t1, t2\rangle$ is defined as the average change for the interval:*

$$\frac{\sum_{t=t1}^{t2} chg(e,t)}{t2-t1} \tag{9.1}$$

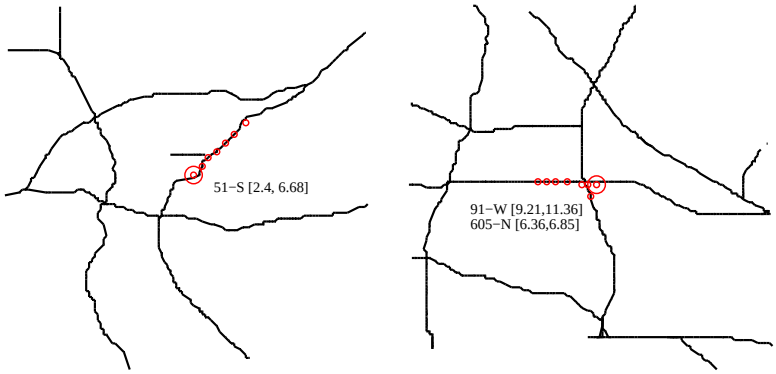
Definition 9.3.6 *Two atypical intervals $c_i = \langle e_i, t1_i, t2_i\rangle$ and $c_j = \langle e_j, t1_j, t2_j\rangle$ are* directly connected*, if (i) e_i, e_j are connected in G, and time intervals $[t1_i, t2_i]$, $[t1_j, t2_j]$ intersect.*

Definition 9.3.7 *There is a* path of change *between two atypical intervals c_i, c_j if they are directly connected, or there exists a sequence of atypical intervals between c_i and c_j and every consecutive pair of change intervals is directly connected.*

Having defined atypical intervals, which are periods of time for which an edge continuously exhibits abnormal speeds, and paths of change which are sequences of connected edges all exhibiting abnormal speeds, during overlapping time intervals, we are ready to define the atypical fragment.

Definition 9.3.8 *An* atypical fragment *N is a set of atypical intervals $N = \{c_1, \ldots, c_k\}$ such that there is a path of change between any two intervals c_i and c_j in N, and every atypical interval in the path is in the set N.*

Intuitively, the atypical fragment is composed of all atypical intervals that are both spatially (connected in G), and temporally (overlapping time) connected. This definition matches the intuitive notion of road anomalies as contiguous road network segments where cars move unusually slowly. Figures 9.4(b), and 9.4(a) presents real examples of atypical fragments (red dots indicate segments with anomalous traffic) that involve multiple road segments, in a single road, and multiple roads respectively.



(a) Single road (b) Multiple road

Figure 9.4: Atypical fragment examples

9.3.4 Atypical Fragment Mining

In this section we introduce an efficient algorithm, AFragMine, to mine atypical fragments from a large traffic database. The key idea of the algorithm is to find edges that exhibit significant changes (by taking advantages of the spatial and temporal localities of the data typically stored in the database), compute time intervals during which the change is continuously observed, and then connect intervals that occur at neighboring edges and that overlap in time. The algorithm requires only a single scan of the database.

The algorithm is efficient because it discovers anomalies only by walking through paths of change, *i.e.*, we merge atypical intervals only if they occur at overlapping times, and the edges where they occur are connected in the road network. This is in contrast to a typical clustering algorithm which would disregard the concept of "connected through change" and spends time in comparing anomalies that can never be merged. Such a methodology of taking advantage of spatial and temporal locality in search, computation, and discovery leads to roughly two orders of magnitude performance gain over a typical clustering-based algorithm, as demonstrated in our experimental evaluation.

Correctness. Every atypical fragment identified by the algorithm is a proper atypical fragment as stated in definition 9.3.8. An atypical fragment requires two conditions (i) that all atypical intervals are connected by a path of change, and (ii) that every atypical interval in a path of change is itself in the fragment. Both of these conditions are explicitly checked on line 17 of the algorithm, as we only add atypical intervals to a fragment if they are directly connected in space and time to a change interval already in the fragment.

Completeness. Every traffic anomaly at an edge that passes the minimum threshold is identified. By definition, line 4 of the algorithm identifies all significant changes in speed. It is possible that the change is merged with other changes, and placed into a large atypical fragment, but it will never be ignored. This is an important property as we guarantee that all significant changes are considered by the algorithm.

Algorithm 9 AFragMine: Atypical Fragment Mining

Input: $G(V, E)$, D: traffic database, δ_r: anomaly threshold
Output: Set of atypical fragments
Method:

1: **for** each edge $e \in E$ **do**
2: **for** each time t **do**
3: $chg = speed(e, t)/f_l(e, t, (d_1, \ldots, d_k))$;
4: **if** $chg > \delta_l$ **then**
5: Add (e, t) to set of change intervals C, merge as necessary;
6: **end if**
7: **end for**
8: **end for**
9: for all i, $c_i.group = unassigned$;
10: group = 1;
11: **for** each change interval c_i **do**
12: **if** $c_i.group == unassigned$ **then**
13: q.push(c_i);
14: **while** q not empty **do**
15: $c = q.pop()$;
16: $c.group = group$;
17: push into q all neighbor intervals of c, such that their time intervals intersect, and their edges are neighbors in G;
18: **end while**
19: **end if**
20: group = group + 1;
21: **end for**
22: return the set of atypical fragments defined by each group;

Compactness. The number of atypical fragments identified by the algorithm is bounded by the number of individual speed anomalies present in the traffic database. In general it is much smaller than this number, and it would only approach this bound when each atypical fragment is composed of one interval of change that contains a single speed anomaly.

Efficiency. Algorithm 9 can be executed in a single scan of the traffic database, steps 1-8 read each entry in the traffic database check for edge anomalies against a pre-computed traffic model. Steps 11-21 perform a walk through edges in G that exhibit intervals of change, each edge interval is visited once, for a run time of $O(|change\ itervals|)$.

9.4 Multi-dimensional Atypical Fragment Overlay

Multi-dimensional anomaly analysis enables the aggregation of atypical fragments in order to discover interesting patterns associated with different anomaly types. For example, one could contrast characteristics, such as duration, length, or formation speed, of traffic jams that appear around highway interchanges and contrast them with those that appear on linear highway segments, or with those with the same topology but located in a different area of the road network. One can also overlay fragments of similar severity, and find contrasting patterns between high and

low severity anomalies. Anomaly overlay provides a new tool for traffic flow analysis, that can be used for automatic anomaly segmentation, and characterization, which in turn can shed new light into problems such as incident detection, bottleneck prediction, and roadway geometric design.

9.4.1 Feature Space

Definition 9.4.1 *An atypical fragment c_i, is described by a feature vector $cf_i = f_{i,1}, \ldots, f_{i,k}$, where $f_{i,j}$ is the value for feature j of atypical fragment c_i.*

An atypical fragment can be described by multiple features which can be categorized into the following two groups:

1. Contextual features describe external conditions present at the time of the anomaly, such as observed *weather conditions* during the anomaly, *properties of the road segments* involved in the fragment (*e.g.*, high occupancy lanes, or road geometry); and the presence of *events* such as reported incidents in the vicinity of the fragment.

2. Intrinsic features are the features extracted directly from the atypical fragments themselves, such as (1) *diameter*, defined as the longest path in the atypical fragment; (2) *radius*, defined as the average path length; (3) *duration*, defined as the total interval of time during which the atypical fragment was present; and (4) *propagation speed*, defined as the average speed at which the atypical fragment propagated along the involved edges. Many other features can be defined depending on the application.

9.4.2 Atypical Fragment Overlay

To discover patterns associated with different types of atypical fragments, we need a mechanism to overlay those fragments. The criteria of overlay depends on the set of features relevant to the analysis. In some cases we may want to merge every atypical fragment appearing in an area of the road network, *e.g.*, to compute general congestion statistics. In others we may want to merge atypical fragments that exhibit similar topologies, *e.g.*, to detect recurrent congestion patterns across different highway configurations such as interchanges, splits, and merges.

Overlay Features. We call the set of atypical fragment features that are relevant for overlay as *overlay features*, and define $d(c_i, c_j)$, the distance between atypical fragments c_i and c_j, as a *function* of the overlay features of the fragments. The definition d will be application dependant. For example, to overlay on topology, the distance may be a graph similarity metric; whereas to overlay on duration and length, the metric may be Euclidean distance.

The set of all possible overlay features forms an *overlay lattice*, with the base element corresponding to an overlay where all features are relevant (*i.e.*, every feature is important in computing the distance between the fragments), whereas the top one corresponding to the one where no feature is relevant (*i.e.*, all fragments can be merged).

Atypical Fragment Clustering. Once we have selected a set of relevant overlay features, the process of merging atypical fragments with similar features can be framed as a typical clustering problem: Given a set of atypical fragments $c_1, \ldots, c_n$, and a distance function $d(c_i, c_j)$ defined over the relevant overlay features, cluster the fragments into disjoint groups with high similarity among fragments inside the group, and high dissimilarity to fragments in other groups.

Our framework can use any typical clustering algorithm for the purpose of atypical fragment overlay. The decision on which algorithms to use depends on the overlay features, distance function, and data characteristics. In our implementation we used both *k*-means [68] and DBScan [28] with good results.

An important application of atypical fragment clustering is characterizing and contrasting of anomaly types. Such process can be accomplished by collecting statistics (or frequent patterns) on the non-overlay features of each cluster and contrasting them with those of another cluster. For example, if we record $(topology, duration, length)$ for each atypical fragment and cluster them according to $duration$, it is possible to contrast long and short clusters in terms of the frequently encountered topologies in each. In the following subsections we highlight several forms of overlay, that are interesting in traffic analysis.

Topology Overlay

Road network topology is an important factor in determining the extent, and severity of traffic anomalies. Different topologies may be associated with changing congestion patterns, *e.g.*, incidents near highway interchanges may generate anomalies that propagate faster, and that last longer than those that occur on single highway segments under the same conditions. Topology can also be associated with features of the road network, such as the presence of bridges, or the number lanes on each segment.

Figure 9.5 presents several typical highway configurations in road networks. Figure 9.5-a represents the merge of two highways, 9.5-b the split of a highway into two, 9.5-c a highway interchange, where traffic can switch highways, and 9.5-d a linear segment of highway with no intersections.

For portions of the road network involving roads other than highways, more complicated patterns can be used for analysis. Such patterns can be provided by transportation engineers, or they can be automatically mined from the road network, by using frequent pattern mining techniques. Given that traffic data is generally only available for highways, we focus on highway topology, but the ideas can be extended to arbitrary road configurations.

Atypical Fragment Simplification. An atypical fragment is a subgraph of the road network, in which all edges exhibit anomalous traffic at connected time intervals. In severe congestions, atypical fragments can involve a large number of edges. To overlay atypical fragments according to topology, it would be unrealistic to only match atypical fragments that are identical, as they may contain a different number of edges, with different extents and durations.

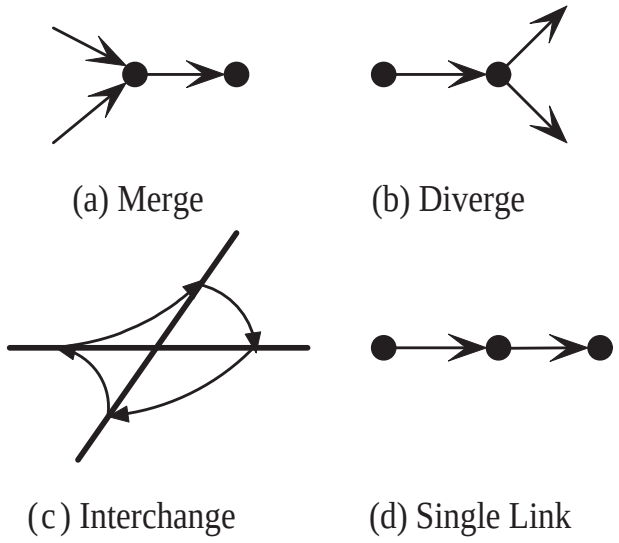


Figure 9.5: Road configurations

That is, if we use strict frequent graph mining to determine the frequency of each topology in a cluster, one can hardly find many completely matching fragments.

We propose approximate match of atypical fragments by *atypical fragment simplification*, which preserves important topological characteristics, as follows: (1) *break each fragment along the involved roads*, *e.g.*, if a fragment involves highway 57 and highway 80, one would group all edges into two groups; (2) *record the topology of each road intersection in the fragment*, and (3) *aggregate all edges on the same road into a single edge*, with duration and length equivalent to the sum of its component edges. Figure 9.6 presents an example of the process: Figure 9.6-a is an atypical fragment that can be simplified into a simpler pattern involving only 3 edges as shown in Figure 9.6-b.

Simplified atypical fragments can be matched according to different overlay criteria: If we care only about topology, patterns can be matched if they present the same set of road intersections. However, if we care about both topology and spatiotemporal characteristics, atypical fragments can be matched if they present the same topology and their corresponding edges exhibit similar spatial and temporal extent.

Severity Overlay

Another interesting overlay criterion is to cluster atypical fragments according to severity, which can be determined by factors such as total duration, total spanned distance, or even a compound measure such as total driver time loss or total vehicle fuel loss due to the fragment. This type of overlay can be easily implemented with the use of a standard distance metric, such as Euclidean distance, or cosine similarity, between points in n-dimensional (defined by the selected severity-related features) space.

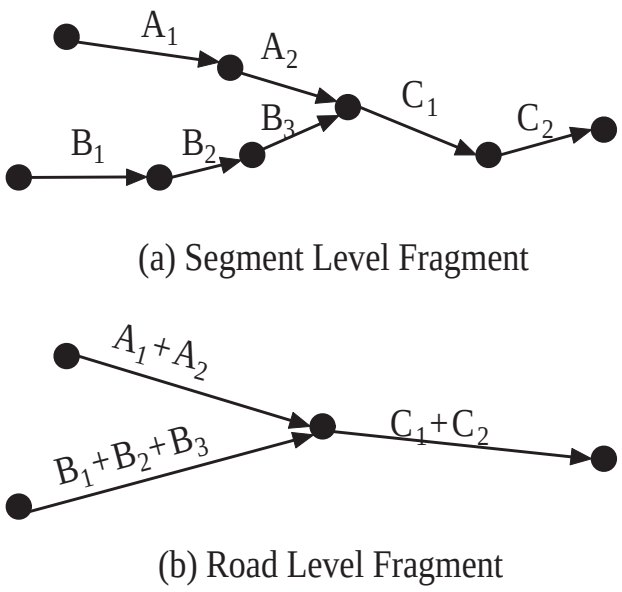


Figure 9.6: Atypical fragment simplification

One application of severity overlay is the identification and contrast of frequent patterns associated with anomalies of different severity levels. In our experiments we found many interesting patterns related to severity. For example, we discovered that anomalies on late Fridays nights (and early Saturdays) presented low severity, but were frequently associated with reported incidents, while high severity anomalies during weekdays were much more likely to be associated with a reported incident than low severity ones.

Spatiotemporal Overlay

Mining recurrent congestion patters is an important problem in traffic engineering, it can be used in incident detection to differentiate between congestions caused by incidents from periodic ones. It can also be used in traffic prediction [57], where bottlenecks (recurrent congestions), are mined to find dependencies and correlations that can be used to predict the formation of traffic jams.

Our framework naturally supports the mining of such patterns. We can overlay atypical fragments according to spatial extent (starting and ending points in a road network), and time extent (starting and ending times relative to the day). That is, two atypical fragments occurring at different days, at around the same road segment milepost, and around the same time of day can be overlayed.

The process to detect recurrent fragments, is as follows: (1) *mine atypical fragments over a period of time*, (2) *cluster atypical fragments according to road level proximity*, and (3) *output the clusters that pass a minimum support threshold* (*e.g.*, the number of points in the cluster). As with other forms of overlay, for each recurrent congestion

cluster, we can mine interesting information, such as the periodicity of the pattern (how often it repeats, and on which days of week, *e.g.*, weekday vs. weekend), its evolution over time, and correlation with contextual features such as weather.

9.4.3 Traffic Anomaly Data Cube

Further extension of the single overlay model can lead to the design and development of a traffic anomaly analysis data cube, which will allow traffic managers to navigate through a series of different clusterings of the data, by drilling down and rolling up according to multidimensional features. For example, one may perform an initial clustering of all atypical fragments along the severity dimensions, and then drill down on a subset of severe clusters to explore more detailed clustering based on weather conditions and road network areas. This will bring the power of OLAP to the analysis and exploration of traffic anomalies and serve as the basis for a multitude of mining tasks such as outlier detection and atypical fragment classification.

Fact Table. In anomaly analysis, we can use the set of contextual features describing an atypical fragment as dimensions of the traffic anomaly cube. Table 9.1 presents an example fact table, and cubing can be done along the dimensions of weather, time of day, and the presence of incidents. Each cell in a cuboid records the set of atypical fragments that share the same values for each contextual dimension.

Measure. Each cell of the anomaly cube contains a measure which, unlike traditional data cubes, is not a scalar value but a clustering of the atypical fragments aggregated at the cell, along a given overlay criterion. Continuing with our example from Table 9.1, the cells $(Weather = Good, Time = Rush\ Hour, Incident = yes)$ and $(Weather = Good, Time = Rush\ Hour, Incident = no)$ would cluster all incidents that occurred during good weather, at rush hour time, and with or without reported incidents, respectively. The aggregated cell $(Weather = Good, Time = Rush\ Hour, Incident = *)$ clusters the union of the atypical fragments in the previous two base cells.

Lemma 9.4.2 *The atypical fragment clustering, performed under a wide range of clustering algorithms, is a holistic measure.*

Proof Sketch. Given a data set D, and a partition of D into disjoint subsets $D_1, \ldots, D_k$, a measure is holistic if there is no bound on the data that each subset D_i needs to provide in order to compute the measure on D. For the clustering case, assume that we have clustered the data in each subset D_i and now need to compute the clustering on the full data set D. There is clearly no fixed sized summary that each D_i can provide in order to cluster the complete data. For a clustering algorithm like DBScan, each cluster would need to provide at least information on border and

outlier points, which has no bound. For a partitioning algorithm like k-means, one cannot derive high-quality cluster centroids of the complete data set by checking only the cluster centroids of the subsets. ■

Weather	Time of Day	Incident	Atypical Fragments
Good	Rush Hour	Yes	$\{C_1, C_2, C_3\}$
Good	Rush Hour	No	$\{C_4, C_5\}$
...	....	...	...
Rain	Rush	Yes	$\{C_6, C_7\}$

Table 9.1: An example fact table

Efficient Computation. Even though clustering is in general a holistic measure, we can still optimize its computation under certain conditions.

Assume that a set of points D is divided into two disjoint subsets D_1 and D_2, and DBScan [27] is used to cluster the points in each subset. Then these two sets of clusters can be used to optimize the computation of the full clustering with the following method: (1) mark all core points in D_1 and D_2 as clustered and assign to them a unique cluster id, (2) mark all border and outlier points as unclustered, and (3) run DBScan again by visiting only non-core points. The algorithm will encounter three cases when visiting non-core points: (i) the points form a new cluster, (ii) the points can be merged into an existing cluster, or (iii) the points can be used to bridge two existing clusters into one. The savings of such strategy is that the core points do not need to be revisited.

Similar optimizations can be explored for other algorithms. For example, a Birch-based micro-clustering method can be used to perform micro-clustering first, which will speed up the processing of the clustering of D_1, D_2, as well as the merged D, for distance-based clustering.

For cube computation, several strategies can be explored. First, one can cluster the cells in the fact table, and use them to compute higher level cells by merging the clusterings of these cells as described before. Alternatively, one can use a more aggressive shared computation strategy in which each cell is computed, not from the base cuboid, but from the smallest cuboid that is below the cell in the cuboid lattice. This strategy has the advantage of reducing computation time, but given the holistic nature of clustering, we may need to store a very significant amount of data for each cell in order to use it to compute higher level aggregates. We believe that the first strategy would be appropriate in most cases to materialize the anomaly with good speedup without significant penalty in space utilization (the size of the annotations is sublinear in the size of the data set).

9.5 Experimental Evaluation

In this section we perform a thorough analysis of our algorithms applied to real traffic data sets for the state of California. All the experiments were conducted on an Intel Pentium 4 processor running at 2.4 GHz, with 1 GB of

RAM. The system ran Linux Kernel 2.6.22, and the code was compiled with gcc 4.1.3.

9.5.1 Experimental Setup

For all our experiments we used real traffic data for the state of California (CA) for March of 2007. The traffic data was obtained from the Freeway Performance Measurement System (PeMS[3]). Road incident information was collected by PeMS from the California Highway Patrol (CHP) web site. And the road network (roads with latitude and longitude of end points) was obtained from the U.S. Census, Tiger line files[4].

The traffic data is a set of readings collected from 10,044 loop detectors placed in 71 freeways crossing the state of California. The format of the traffic data is $(vds\_id, time, occ_1, flow_1, \ldots, occ_k, flow_k)$, where vds_id is a unique identifier for the set of loop detectors placed on each lane of the freeway at a given location, $time$ is a 30 second interval, occ_i is the occupancy for lane i, and $flow_i$ is the flow for lane i. Additionally, for each detector we know its location on the freeway, and its type, which can be a main line detector, on- or off-ramp detector, or a freeway to freeway detector. After importing the traffic data we removed inconsistent readings (impossible combination of flow and occupancy), estimated the speed from occupancy and flow, interpolated missing readings, and smoothed the estimated speed with a smoothing window of size 5 to reduce the effects of sensor noise.

We constructed a road network graph for the subset of road segments that were monitored by sensors. Each edge of the graph is associated with a sensor, and edges are connected if they are on the same freeway, or if they are freeway intersection points and they are close.

9.5.2 Efficiency

In this set of experiments we evaluate (1) the scalability of AFragMine, (2) the compression power of the algorithm, and (3) the benefits of shared computation of atypical fragment clusters in a cubing environment.

Speedup of Atypical Fragment Mining

In this experiment we compare the efficiency of the atypical fragment mining algorithm compared to a clustering algorithm. We implemented an agglomerative clustering algorithm that takes as input all the change intervals, *i.e.*, $\langle t1, t2, eid \rangle$, where $[t1, t2]$ is the time interval during which edge eid had anomalous speed. and merges them into clusters as long as their time intervals are close and spatial distance between edges is small.

Figure 9.7 presents the runtime of our algorithm AFragMine vs. the clustering algorithm *cluster*. The runtimes do not include time to load the data from disk, as it is constant and independent of the algorithm. We see that our method

[3] `https://pems.eecs.berkeley.edu/`
[4] `http://www.census.gov/geo/www/tiger/tgrcd108/tgr108cd.html`

is around two orders of magnitude faster, even though it has to account for mining of intervals of change, whereas the clustering algorithm already gets pre-processed data. We tried to run the clustering algorithm on raw data but it took tens of minutes to finish even on the smallest data sets. In addition, the clustering algorithm has the difficulty of defining a relevant distance measure for traffic anomalies. The reason for the slow performance of a typical clustering algorithm is that it visits each change interval multiple times, trying to build successively larger clusters. In our case, each change interval is visited only once, as we expand it to form atypical fragments.

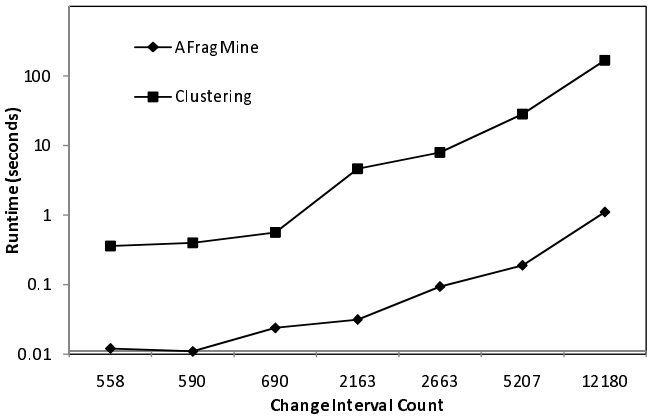


Figure 9.7: Change intervals vs. Runtime

Compression Power of AFragMine

Figure 9.8 compares the number of atypical fragments discovered by AFragMine with the total number of change intervals, defined as in the previous experiment. We see that AFragMine compresses the number of discovered anomalies, without loss of information, by almost two orders of magnitude. Such compression has two main benefits: (1) a richer semantic anomaly model that captures important characteristics, such as total duration and extent, or propagation speed, which are lost in the raw anomaly data; and (2) reduction in data size, which is vital for the efficient execution of higher level algorithms such as anomaly overlay, and especially anomaly cubing, which exhibits an exponential speedup for linear reductions on the size of the fact table.

Benefits of Shared Computation

In this experiment we compare the benefits of shared computation of atypical fragment clusters from existing clusters vs. the computation of such clusters from scratch. We take a data set D and randomly divide it into two subsets D_1 and D_2 of roughly the same size. We cluster D_1 and D_2 independently, using DBScan [28], and compute the savings of clustering D starting from these two clusterings vs that of starting from the original data. In Figure 9.9 we present

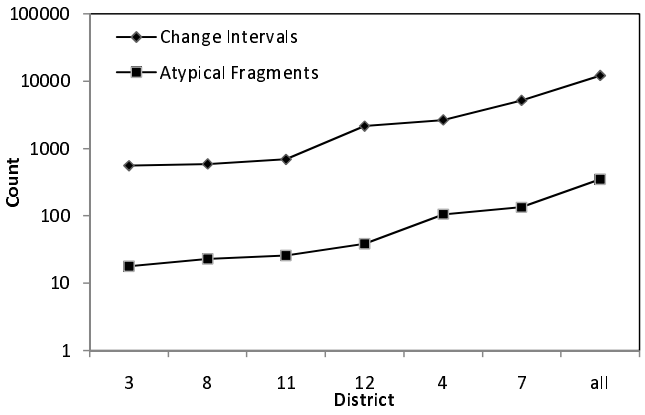


Figure 9.8: Compression

two cases: (1) *full overlap* where the points for each cluster in D reside in both subsets, and (2) *no overlap* in which the points for each cluster in D reside only at a single subset. We vary the eps parameter to change the number of core and border points. At low levels of eps we see that with *full overlap* we have to process about 70% of the data points, and with *no overlap* we process only about 30%. As we increase eps, more points become core points, and we increase the savings. In a real system, the savings will lie in between the two extreme cases of *full overlap* and *no overlap* presented in the experiment, and we can see that even in the most difficult environment we can still obtain performance gains from shared computation. For lack of space, we do not present savings in microclustering-based method, which is also a viable shared computation model.

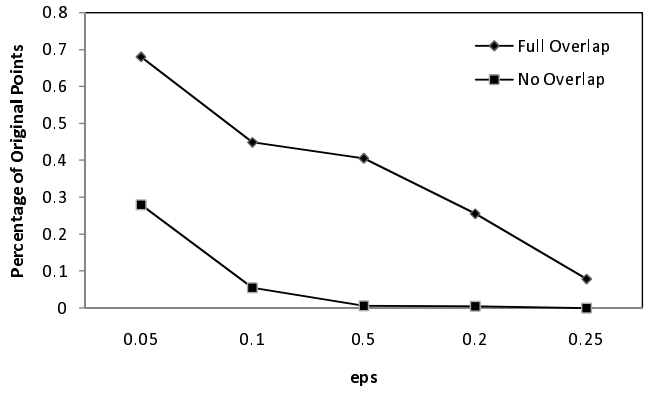


Figure 9.9: Shared computation savings

9.5.3 Qualitative Results

To illustrate the power of the anomaly mining framework, we present a few examples of multi-dimensional anomaly overlay over real traffic data collected from the month of March of 2007 for the Bay area district in California. They illustrate the importance of OLAP style queries in navigating large traffic data sets. For these experiments we used the anomaly threshold of 0.3, which is the percentage of change in speed with respect to normal before it is considered anomalous, and considered only atypical fragments that involved at least two loop sensors.

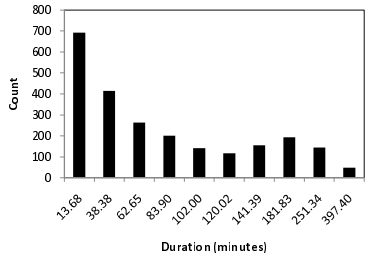


Figure 9.10: Severity overlay complete data

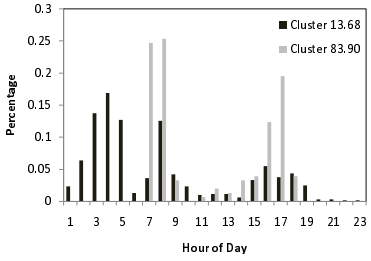


Figure 9.11: Complete data: Contrast by hour of day

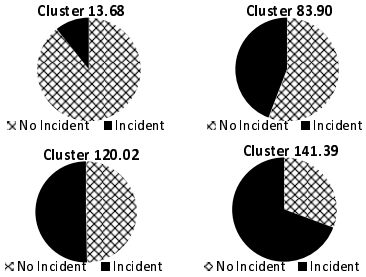


Figure 9.12: Complete data: Contrast by incident

Figure 9.10 presents the results of overlaying atypical fragments on severity, using the criteria of total atypical fragment duration. Each value on the x-axis corresponds to a cluster, and the y-axis is the number of atypical fragments in the cluster. We observe that more than 50% of the atypical fragments have a duration of less than 40 minutes. In Figure 9.11 we contrast two clusters, one with average duration of 13.68 minutes vs. one with average duration of 83.9 minutes. We compare them in terms of the time of day when elements on each cluster appear. We see that for the short cluster the distribution is more even during the day, whereas for the long cluster the anomalies appear almost exclusively during rush hour (7am - 9am, 4pm - 6pm). In Figure 9.12, we contrast the percentage of atypical fragments that are associated with a reported incident (obtained from CHP), we compare 4 clusters of different durations. The figure shows that in this case, clusters with higher average duration have a higher percentage of incidents.

In Figure 9.13 we present the clustering of atypical fragments, after we drill down from the clustering in Figure 9.10 along the time dimension to the hour level, and have selected only the hours of 7am to 8pm. Figure 9.14 is also a drill down into the hour level, but we select hours from 12am to 5am. By contrasting Figures 9.13 and 9.14 we observe that daytime anomalies are not only more numerous, but also have higher severity (in terms of duration) than night time anomalies. By drilling down to the hour level, we can analyze the contribution of different times of day to the aggregate anomaly patters for the day. Figure 9.15 presents yet another navigation path from the clustering in Figure 9.10 to the cuboid with anomalies for only Saturday. The figure contrasts a long and a short cluster within the cuboid, along the hour of the day dimension. We see an interesting pattern, severe anomalies tend to occur on Saturday

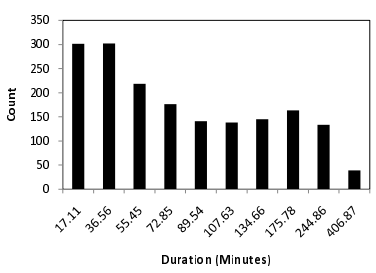


Figure 9.13: Severity overlay 7-8pm

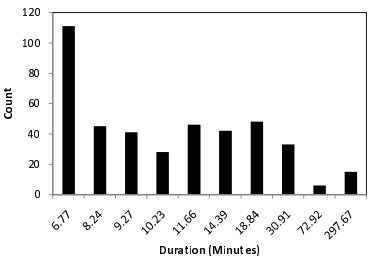


Figure 9.14: Severity overlay 0-5am

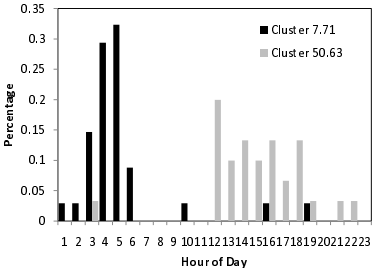


Figure 9.15: Saturday: Contrast by hour of day

afternoon, whereas shorter anomalies (although more frequent) tend to occur between 12am and 6am, likely the time when people go back home after a night in town, and when incidents are likely to occur, but with short duration given low highway occupancy.

Figure 9.16 presents the results of spatiotemporal overlay of atypical fragments. We aggregate all atypical fragments that occur in the same freeway, at close mileposts, and around the same time of day. Such overlay is useful to discover recurrent anomalies (bottlenecks), which in turn can be used for traffic prediction [57], traffic management, and road design. We list the roads and the number of atypical fragments that appear at least 15 times during the month of March of 2007. With recurrent anomalies we can also drill down along different dimensions: in our study we drilled down on day of the week and observed some interesting patterns. For example, we discovered that on freeway 17-N there is a bottleneck that forms every weekday, during the morning, between mileposts 16.5 and 20, and lasts for 67 minutes. On the same freeway we observed a different bottleneck that forms in the afternoon, between mileposts 18 and 20, that only appears on weekends, and that lasts for 16 minutes.

Overlay results on topology show that the single-link topology turns out to be dominating. But, we believe that in denser road configurations, such as city roads, anomalies involving more complex topologies would increase frequency. Unfortunately, at this point, sensor data for city roads is generally unavailable.

9.5.4 Anomaly Threshold Tuning

In this set of experiments we explore the behavior of AFragMine under different parameter settings. For these experiments we use the traffic database for a single day 01/01/2007, for the times between 8am to 10am. Figure 9.17 presents the number of anomalies identified by the algorithm by varying the anomaly threshold parameter. This experiment suggests that anomalies on real road networks are not very sensitive to the anomaly threshold parameter, and that using a value of around 30% is enough to discover most interesting anomalies. The reason is that most anomalies present significant deviation from normal traffic parameters, and when we look at the full extent of each anomaly, it is

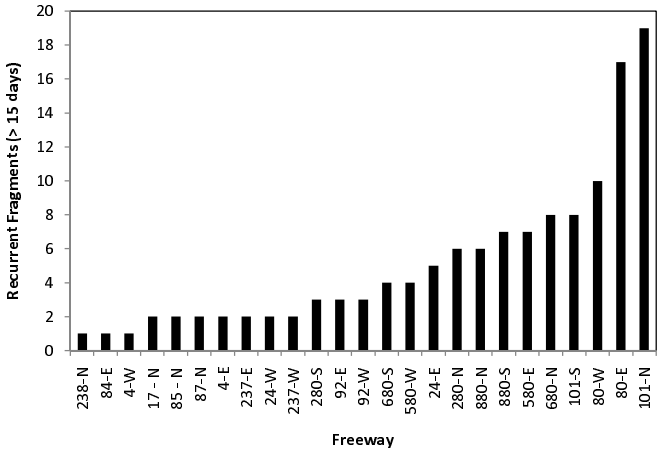


Figure 9.16: Recurrent anomalies

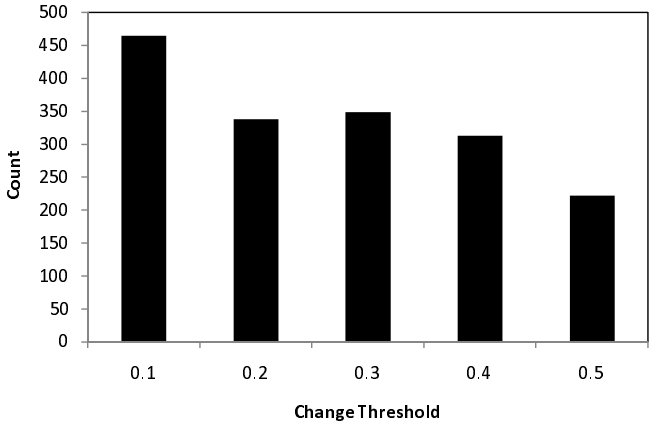


Figure 9.17: Anomaly threshold vs. Anomaly count

easy to identify at least a portion of the anomaly which passes the threshold.

9.6 Summary

In this chapter we introduced the concept of the atypical fragment as a novel representation of traffic anomalies. The atypical fragment departs from traditional traffic anomaly definitions in that it groups a large number of anomalies connected in time and space into a single unit.

We proposed an efficient algorithm, AFragMine, for mining atypical fragments. We showed that unlike anomaly clustering we can compute atypical fragments in linear time. More importantly, our algorithm can be applied to sensor data for a large real road network, and find a very compact set of anomalies that is orders of magnitude smaller that

the complete set of raw anomalies. Such compression not only is vital in improving semantic understanding of traffic anomalies, but also enables efficient algorithms such as atypical fragment overlay and cubing.

We presented a novel anomaly overlay model, that can be used to aggregate anomalies according to different criteria such as, topology, severity, and spatiotemporal characteristics. Such overlay can be used to identify underlaying characteristics associated with certain types of anomalies that are only observable on aggregated data. We showed that finding recurrent bottlenecks, a very important problem in traffic engineering, is just a matter of varying the overlay criteria in the framework.

Finally, we introduced a traffic anomaly cube model, in which each cube cell contains an overlay of traffic anomalies according to a set of contextual dimensions. We showed that such model is useful in OLAP of anomaly data, to discover information on recurrent patterns, outliers, and trends. Several shared clustering methods, that improve cube computation time, where presented. As a proof of concept, we implemented the complete framework and successfully applied it to the analysis of real traffic data from California.

Chapter 10

Conclusions and Future Directions

This chapter summarizes the moving object framework proposed in this dissertation and discusses future directions for research motivated by the results obtained in the study, and current technology trends.

10.1 Conclusions

The continuous growth of automatic object identification technologies has brought tremendous improvements to applications such as supply chain management, asset tracking, and traffic monitoring. But it has also created important data management challenges. The enormous size of the datasets, their complex spatio-temporal characteristics, and the wide range of queries and mining tasks are all issues that need to be addressed in order to realize the full potential of the technology.

In this dissertation we have proposed a comprehensive framework for the management and mining of massive moving object datasets. We addressed the following challenges.

Construction of an RFID data warehouse capable of handing massive data sets, while preserving their path structure. Compression was achieved based on: (1) removing redundancy due to periodic readings, and (2) registering object movements at the group level. The warehouse design is centered around the idea of the *movement graph*, which in turn is used for data partitioning, improved compression, and faster query processing.

Cleaning of massive RFID datasets. We departed from existing techniques that focus on a single cleaning method and proposed the concept of the cleaning plan, which applies different data cleaning methods depending on the context in which the data is generated. We showed that the cleaning plan significantly reduces cleaning costs, while providing higher accuracy than any cleaning method used in isolation. The cleaning plan, is to the best of our knowledge, the first study that addresses the problem of cleaning scalability in large RFID implementations.

Mining of interesting object flow patterns at multiple levels of abstraction. We introduced the FlowGraph, and FlowCubes as a means to summarize and discover spatial and temporal correlations in flow trends. The model can be used to discover major flow trends, significant exceptions to the trend, and important spatio-temporal correlations present in massive RFID datasets.

In the context of road network monitoring through RFID technology and related sensors, we proposed two mining methods. (1) Mining route recommendations from vehicular traffic data. We proposed a data centric route recommendation algorithm, that observed driving behavior to suggest routes that are not only fast but also popular. We showed that such data centric, model can capture a variety of factors, that would be very hard to explicitly incorporate in a graph-centric route planning model. (2) Mining traffic anomalies in road networks. We proposed the Atypical Fragment concept, as a concise representation of traffic anomalies, and proposed an anomaly overlay framework useful to explain congestion patterns based on regularities across traffic anomalies.

10.2 Future Directions

Recent advances in object tracking technologies such as RFID, data collection systems in road networks, and cell phone tracking through GPS and signal triangulation have opened new frontiers in research, industry, and government. I would like to study human and object movements, to discover fundamental patterns, that will help us architect a world that is aware of, and adapts to our behavior and needs. Towards this goal I have developed several methods in the areas of warehousing, mining, and cleaning of moving object data, but there is still a very fertile ground for significant new theoretic and practical developments that will have a positive direct impact on society in the near and long terms. My research plans in the near term are illustrated as follows:

10.2.1 Mining Human Movements

In the near future it will be possible to track human movements in order to improve the design of buildings, stores, or public spaces. For example, we could provide building occupants with RFID enabled badges, and place readers throughout the building, or in a store, we could place an RFID reader in each shopping cart and track the customer trajectory and items picked throughout the store. Movement sessions can be represented by a sequence of tuples of the form $(session\_id, time, location, action\_set)$, where we record the set of actions taken by the person at a given location and time. With this information several problems can be formulated.

Mining for optimization of building layouts: Good building design can be aided by movement data, the way in which people move and interact with building facilities can be used to improve its layout. Given a set of movement sessions, we can arrange facilities to maximize convenience, or in a store setting rearrange products as to maximize sales. We could for example take the collective movement of all building occupants and propose a layout that minimizes aggregate trajectory length; or in a store we could find an arrangement of products that by placing needed, and impulse products at strategic locations increases customer satisfaction and total purchases. Mathematical programming alone can not be used to tackle such problems, as the number of movement sessions can be in the hundreds of thousands,

the set of constraints can be enormous, and the function to optimize can be hard to define precisely and non-convex. Data mining can be a valuable tool to discover important movement/purchasing patterns that can be used as a building block in solving the problem.

Integration of movement patterns in social network analysis: Significant research attention has been given to understanding the networks of social relationships between individuals or organizations. Social network analysis helps in the understanding of topology, formation, and evolution of such networks. We can for example rank participants according to their importance or centrality in the network, predict the formation of new relationships, and evaluate the effectiveness of different topologies in conveying information. I believe that in the near future we will be able to incorporate information on physical actions and movements of network participants into the analysis. I believe that the way we move, and the actions we take in the real world, have a deep connection to the way we relate to others in society. We can for example, look at shared activities, or track trajectories that frequently intersect, to identify hidden social links. I would like work on the problem of mining spatio-temporal patterns relevant to the understanding of social relationships, and explore its applications into areas such as, location-aware services, and national security.

Movement-based classification: I believe that movement patterns of customers in a store, or visitors to a building, can provide valuable clues as to the needs and intentions of such persons. For example, we could, identify patterns associated with purchase intent, shoplifting propensity, or confusion level. Purchase intent is an important measure to improve sales force utilization, we could determine if a customer is about to make a significant purchase given the set of articles in the cart, the set of locations visited, and the time spent at each. Preventing shoplifting is another important application, it is possible to use trajectory and buying behavior to find suspicious customers, we could for example find customers that switch labels of expensive and cheap products, by checking the locations visited by the customer, the time spent at each, and the product finally scanned at the checkout counter. Lost visitors to a building can identified and helped, if we look patters such as frequent erratic trajectories.

Data mining for advanced transportation engineering: In modern road networks vehicles are tracked as they move through highways and city roads. Such data can be used to improve existing traffic engineering techniques for automatic incident detection, traffic management, and road upgrade planning. The detailed set of movement sessions in a city provides important clues on hot routes, traffic anomalies, and design inefficiencies. I would like to work on the mining of traffic data sets to discover interesting driving and movement patterns that can be used to improve our road networks, make them safer, and reduce driving times.

10.2.2 Mining and Managing the Internet of Things

Technologies such as RFID, and sensor networks are dissolving into the fabric of things. Clothing, car parts, medicines, bridges, and even trees are becoming nodes in a global network of everyday objects. Some predict that in the future we will Google for our lost keys, bridge components will report on structural problems, and our bags will remind us that we forgot to pack our glasses. This new Internet will be gigantic, currently we live in a world of one or at most a few computers per person, but in the future each individual will posses thousands of networked devices. We need to develop new data management and mining technologies to effectively store and analyze the massive data that such a network will generate.

Mining object networks: I would like to mine networks of everyday objects to discover the patterns in which they interact. We could track the communications of smart components in an airplane to identify recurrent patterns that indicate component failure, or patterns that indicate the need for an upgrade. Or, in a world where medicines and patients are nodes in a network, we could analyze their interactions to discover if a patient is taking the right treatment, or to mine classification and regression models that indicate the effectiveness of a drug in treating a certain disease for a given demographic group.

Data warehousing and management: The massive scale of data residing in the Internet of things is orders magnitude larger than the current Internet size. This is a vexing problem, that requires rethinking many existing database, and data warehousing technologies. Existing facilities can handle tera- or even peta-byte size data sets, but such sizes, can be reached in a single day of object interaction data. I would like to extend my work on compressing and warehousing massive RFID data sets to handle object networks, and develop new techniques to warehouse streaming object movement and interaction data. Hardware and communication technologies have connected previously isolated objects into a massive network, but as portals and search engines gave life to the Internet, we need to develop new data management technologies that will allow us to explore and analyze these massive networks and truly convert them into an Internet of Things.

References

[1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 506–521, Bombay, India, Sept. 1996.

[2] R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. In *Proc. Sixth Int. Conf. on Extending Database Technology*, pages 469–483, 1998.

[3] R. Agrawal and R. Srikant. Fast algorithm for mining association rules in large databases. In *Research Report RJ 9839*, IBM Almaden Research Center, San Jose, CA, June 1994.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering (ICDE'95)*, pages 3–14, Taipei, Taiwan, Mar. 1995.

[5] The application level events (ALE) specification. Specification, EPCglobal Inc, 2003.

[6] Anjali Awasthi, Yves Lechevallier, Michael Parent, and Jean-Marie Proth. Rule based prediction of fastest paths on urban networks. In *Proc. 2005 Intelligent Transportation Systems (ITS05)*, pages 978–983, 2005.

[7] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 359–370, Philadelphia, PA, June 1999.

[8] L.D. Bloomberg, V.W. Bacon, and A.D. May. Freeway detector data analysis: smart corridor simulation evaluation. Technical report, 1993.

[9] Arnold P. Boedihardjo and Chang-Tien Lu. AOID: Adaptive on-line incident detection system. In *Intelligent Trasportation Systems Conference*, pages 858–863, Toronto, Canada, September 2006.

[10] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[11] Thomas Brinkhoff. Network-based generator of moving objects. Technical report, IAPG, `http://www.fh-oow.de/institute/iapg/personen/brinkhoff/generator/`.

[12] David L. Brock. The electronic product code (EPC): A namig scheme for physical objects. White paper, MIT Auto-ID Center, 2001.

[13] J. Brusey, C. Floerkemeier, M. Harrison, and M. Fletcher. Reasoning about uncertainty in location identification with RFID. In *In Workshop on Reasoning with Uncertainty in Robotics at IJCAI-2003*, 2003.

[14] I. V. Cadez, S. Gaffney, and P. Smyth. A general probabilistic framework for clustering individuals and objects. In *Proc. 2000 Int. Conf. Knowledge Discovery in Databases (KDD'00)*, pages 140 – 149, Boston, MA, Aug. 2000.

[15] Rafael C. Carrasco and Jose Oncina. Learning stochastic regular grammars by means of state mergin method. In *Second International Colloquium on Grammatical Inference (ICGI'94)*, pages 139–150, 1994.

[16] E.C.P. Chang. Fuzzy systems based automatic freeway incident detection. In *Proc. IEEE Internation al COnference on Systems, Man, and Cybernetics*, pages 1727–1733, San Antonio, Texas, 1994.

[17] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26:65–74, 1997.

[18] C. Chen, J. Kwon, A. Skabardonis, and P. Varaiya. Detecting errors and imputing missing data for single loop surveillance systems. In *Transportation Research Record*, pages 160–167, 2003.

[19] R.L. Cheu, D Srinivasan, and T.T. Eng. Support vector machine models for freeway incident detection. In *Intelligent Trasportation Systems Conference*, pages 238–243, 2003.

[20] Yu-Li Chou, H. Edwin Romeijn, and Robert L. Smith. Approximating shortest paths in large-scale networks with an application to intelligent transportation sytems. 10:163–179, 1998.

[21] R.K. Chung. *Spectral Graph Theory*, volume 92. CBMS Regional Conference Series in Methematics, 1997.

[22] J.F. Collins, C.M. Hopkins, and J.A. Martin. Automatic incident detection - trrl algorithms hiocc and patreg. *TRRL Supplementary Report*, 526, 1979.

[23] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Highway hierarchies star. In *9th DIMACS Implementation Challenge*, 2006.

[24] Camil Demetrescu, Stefano Emiliozzi, and Giuseppe F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. In *15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, pages 369–378, New Orleans, Louisiana, 2004.

[25] E.W. Dijkstra. A note on two problems in connexion with graphs. 1:269–271, 1959.

[26] Tapio Elomaa and Juho Rousu. General and efficient multisplitting of numerical attributes. 36(3):201–244, 1999.

[27] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu. Incremental clustering for mining in a data warehousing environment. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, New York, NY, Aug. 1998.

[28] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases. In *Proc. 1996 Int. Conf. Knowledge Discovery and Data Mining (KDD'96)*, pages 226–231, Portland, OR, Aug. 1996.

[29] K. P. Fishkin, B. Jiang, M. Philipose, and S. Roy. I sense a disturbance in the force: Unobstrusive detection of interactions with rfid-tagged objects. In *Proc. 2004 Int. Conf. on Ubiquitous Computing*.

[30] C. Floerkemeier, D. Anarkat, T. Osinski, and M. Harrison. PML core specification 1.0. White paper, Auto-ID Center, 2003.

[31] C. Floerkemeier and M. Lampe. Issues with rfid usage in ubiquitous computing applications. In *Pervasive Computing (PERVASIVE) Lecture notes in Compute Science*.

[32] Robert W. Floyd. Algorithm 97: Shortest path. 5(6):345, 1962.

[33] L.C. Freeman. A set of measures of centrality based on betweenness. 40:35–41, 1977.

[34] Daniele Frigioni, Mario Ioffreda, Umberto Nanni, and Giulio Pasquale. Experimental analysis of dynamic algorithms for the single-source shortest-path problem. *ACM Journal of Experimental Algorithms*, 3, 1998.

[35] L. Fu, D. Sun, and L. R. Rilett. Heuristic shortest path algorithms for transportation applications: state of the art. *Computers and Operation Research*, 33(11):3324–3343, 2006.

[36] S. Gaffney and P. Smyth. Trajectory clustering with mixtures of regression models. In *Proc. 1999 Int. Conf. Knowledge Discovery and Data Mining (KDD'99)*, pages 63–72, 1999.

[37] Nicholas J. Garber and Lester A. Hoel. *Traffic and Hightway Engineering*. Thompson Engineering, 2002.

[38] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest: A framework for fast decision tree construction of large datasets. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 416–427, New York, NY, Aug. 1998.

[39] EPC radio-frequency identity protocols class-1 generation-2 UHF RFID procol for communications at 860 mhz - 960 mhz. Specification, EPCglobal Inc, 2003.

[40] H. Gonzalez, J. Han, and X. Li. Flowcube: Constructuing RFID flowcubes for multi-dimensional analysis of commodity flows. In *Proc. 2006 Int. Conf. Very Large Data Bases (VLDB'06)*, Seoul, Korea, Sept. 2006.

[41] H. Gonzalez, J. Han, and X. Li. Mining compressed commodity workflows from massive rfid data sets. In *Proc. 2006 Conf. On Information and Knowledge Management (CIKM'06)*, Virginia, November 2006.

[42] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and analysis of massive RFID data sets. In *Proc. 2006 Int. Conf. Data Engineering (ICDE'06)*, Atlanta, Georgia, April 2006.

[43] H. Gonzalez, J. Han, X. Li, M. Myslinska, and J.P. Sondag. Adaptive fastest path computation on a road network: A traffic mining approach. In *In Proc. 2007 Int. Conf. on Very Large Data Bases (VLDB'07)*, pages 794–805, Vienna, Austria, September 2007.

[44] H. Gonzalez, J. Han, Y. Ouyang, and S. Seith. Multi-dimensional mining of traffic anomalies on massive road networks. In *In Submission*.

[45] H. Gonzalez, J. Han, and X. Shen. Cost-conscious cleaning of massive RFID data sets. In *Proc. 2007 Int. Conf. Data Engineering (ICDE'07)*, Istanbul, Turkey, April 2007.

[46] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–54, 1997.

[47] M. Gross and R. Feldman. National transportation statistics 1997. *Technical Report DOT-VNTSC-BTS-96-4*, 1996.

[48] Angshuman Guin. *An Incident Detection Algorithm Based on Discrete State Propagation Model of Traffic Flow*. Ph.D. Thesis, Georgia Institute of Technology, 2004.

[49] J. Han and J. Pei. Mining frequent patterns by pattern-growth: Methodology and implications. *SIGKDD Explorations (Special Issue on Scalable Data Mining Algorithms)*, 2:14–20, 2000.

[50] J. Han, N. Stefanovic, and K. Koperski. Selective materialization: An efficient method for spatial data cube construction. In *Proc. 1998 Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD'98)*, Melbourne, Australia, April 1998.

[51] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, pages 211–218, Maebashi, Japan, Dec. 2002.

[52] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96)*, pages 205–216, Montreal, Canada, June 1996.

[53] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. 4:100–107, 1968.

[54] J. A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, 1975.

[55] 13.56 mhz ism band class 1 radio frequency identification tag interface specification. Technical report, MIT Auto-ID Center, 2003.

[56] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Reading, Massachusetts, 1979.

[57] E. Horvitz, J. Apacible, R. Sarin, and L. Liao. Prediction, expectation, and suprise: Methods, designs, and study of a deployed traffic forecasting service. In *Proc. 2005 Conf. on Uncertainty in Artificial Intelligence (UAI'05)*, Edinburgh, Scotland, July 2005.

[58] Shawn R. Jeffery, Minos Garofalakis, and Michael J. Franklin. Adaptive cleaning for RFID data streams. In *Proc. 2006 Int. Conf. on Very Large Data Bases (VLDB'06)*, Seoul, Korea, Sept. 2006.

[59] S. R. Jeffrey, G. Alonso, M.J. Franklin, W. Hong, and J. Widom. A pipelined framework for online cleaning of sensor data streams. In *Proc. 2006 Int. Conf. on Data Engineering (ICDE'06)*, Atlanta, Georgia, April 2006.

[60] X. Jin, R.L. Cheu, and D. Srinivasan. Development and adaptation of constructive probabilistic neural network in freeway incident detection. *Transportation Research. Part C: Emerging Technologies*, 10:121–147, 2002.

[61] Ning Jing, Yun-Wu Huang, and Elke A. Rundensteiner. Hierarchical optimization of optimal path finding for transportaton applications. In *Proc. 1996 Int. Conf. Information and Knowledge Management (CIKM'96)*, pages 261–268, 1996.

[62] Sungwon Jung and Sakti Pramanik. HiTi graph model of topographical road maps in navigation systems. In *Proc. 1996 Int. Conf. on Data Engineering (ICDE'96)*, pages 76–84, 1996.

[63] Evangelos Kanoulas, Yang Du, Tian Xia, and Donghui ZXhang. Finding fastest paths on a road network with speed patterns. In *Proc. 2006 Int. Conf. on Data Engineering (ICDE'06)*, Atlanta, GA, April 2006.

[64] J.H. Kell, I.J. Fullerton, and M.K. Mills. Traffic detector handbook. Technical report, 1990.

[65] E. J. Keogh and M. J. Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In *Proc. 1998 Int. Conf. Knowledge Discovery and Data Mining (KDD'98)*, pages 239–243, New York, NY, Aug. 1998.

[66] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *IEEE Symposium on Foundations of COmputer Science*, pages 81–91, 1999.

[67] K. Koperski and J. Han. Discovery of spatial association rules in geographic information databases. In *Proc. 1995 Int. Symp. Large Spatial Databases (SSD'95)*, pages 47–66, Portland, Maine, Aug. 1995.

[68] J. MacQueen. Some methods for classification and analysis of multivariate observations. *Proc. 5th Berkeley Symp. Math. Stat. Prob.*, 1:281–297, 1967.

[69] M. Mealling. Auto-ID object name service (ONS) 1.0. Working draft, Auto-ID Center, 2003.

[70] S. Pallottino and M.G. Scutella. Shortest path algorithms in transportation models: Classical and innovative aspects. pages 245–281, 1998.

[71] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *Proc. 2001 Int. Symp. on Spatial and Temporal Databases (SSTD'01)*, Redondo Beach, CA, July 2001.

[72] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. 7th Int. Conf. Database Theory (ICDT'99)*, pages 398–416, Jerusalem, Israel, Jan. 1999.

[73] H.J. Payne and S.C. Tignor. A freeway incident-detection algorithms based on decision trees with states. *Transportation Research Record*, 682:30–37, 1978.

[74] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 215–224, Heidelberg, Germany, April 2001.

[75] Karl Frazier Petty. *Incidents on the Freeway: Detection and Management*. Ph.D. Thesis, University of California at Berkeley, 1993.

[76] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[77] J.R. Quinlan. Improved use of continuous attributes in c4.5. 4:77–90, 1996.

[78] Lawrence Rabiner. A tutorial on hidden markov models and selected applications in speech recognition, 1989.

[79] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach. Second Edition.* Prentice-Hall, 2003.

[80] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact path queries. pages 568–579, 2005.

[81] Sanjay Sarma, David L. Brock, and Kevin Ashton. The networked physical world. White paper, MIT Auto-ID Center, 2000.

[82] Auto-id savant specification 1.0 (work in progress). White paper, Auto-ID Center, 2003.

[83] S. Shekhar and Y. Huang. Discovering spatial co-location patterns: A summary of results. In *Proc. 2001 Int. Symp. Spatial and Temporal Databases (SSTD'01)*, Redondo Beach, CA, July 2001.

[84] A. Shukla, P. M. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 488–499, New York, NY, Aug. 1998.

[85] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95)*, pages 407–419, Zurich, Switzerland, Sept. 1995.

[86] N. Stefanovic, J. Han, and K. Koperski. Object-based selective materialization for efficient implementation of spatial data cubes. *IEEE Transactions on Knowledge and Data Engineering*, 12:938–958, 2000.

[87] Franck Thollard, Pierre Dupont, and Colin dela Higuera. Probabilistic dfa inference using kullback-leibler divergence and minimality. In *Proc. Int. Conf. on Machine Learning (ICML'00)*, pages 975–982, 2000.

[88] Peter D. Turney. Cost-sensitive classification: Empirical evaluation of a hybrid genetic decision tree induction algorithm. *Journal of Artificial Intelligence Research*, 2:369–409, 1995.

[89] W.M.P. van der Aalst and A.J.M.M Weijters. Process mining: A research agenda. In *Computers in Industry*, pages 231–244, 2004.

[90] E. van Zwet, C. Chen, Z. Jia, and J. Kwon. A statistical method for estimating velocity from single loop detectors. In *Transportation Research Record*, 2001.

[91] J. Wang, J. Han, and J. Pei. CLOSET+: Searching for the best strategies for mining frequent closed itemsets. In *Proc. 2003 ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD'03)*, pages 236–245, Washington, DC, Aug. 2003.

[92] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Proc. 2003 Int. Conf. Very Large Data Bases (VLDB'03)*, pages 476–487, Berlin, Germany, Sept. 2003.

[93] J. S. Yoo and S. Shekhar. A partial join approach to mining co-location patterns. In *Proc. 2004 Int. Workshop on Geograph ic Information Systems (GIS'04)*, Washington, D.C., Nov. 2004.

[94] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 159–170, Tucson, AZ, May 1997.

[95] George K. Zipf. *Human Behaviour and the Principle of Least-Effort*. Addison-Wesley: Cambridge, MA, 1949.

Author's Biography

Hector Gonzalez finished his B.S. degree in Computer Science from Universidad Eafit, where he graduated first in class. He completed an M.B.A. at Harvard Business School, graduating with distinction in 1999. In 2003 he completed his M.S. in computer science from University of Illinois at Urbana-Champaign. Hector completed his Ph.D. at University of Illinois at Urbana-Champaign in 2008. He conducted research on data mining, data warehousing and database systems, supervised by Professor Jiawei Han. His research, has resulted in publications at major conferences in database systems and data mining, such as VLDB, ICDE, CIKM, and SDM. One of his research papers, "Warehousing and Analysis of Massive RFID Data Sets", received the Best Student Paper award at ICDE'06.